

KATHOLIEKE UNIVERSITEIT LEUVEN FACULTEIT INGENIEURSWETENSCHAPPEN DEPARTEMENT ELEKTROTECHNIEK–ESAT Kasteelpark Arenberg 10, 3001 Leuven-Heverlee

Cryptanalysis and Design of Stream Ciphers

Promotor: Prof. Dr. ir. Bart Preneel Proefschrift voorgedragen tot het behalen van het doctoraat in de ingenieurswetenschappen door

Hongjun Wu 伍 宏 军

July 2008



KATHOLIEKE UNIVERSITEIT LEUVEN FACULTEIT INGENIEURSWETENSCHAPPEN DEPARTEMENT ELEKTROTECHNIEK–ESAT Kasteelpark Arenberg 10, 3001 Leuven-Heverlee

Cryptanalysis and Design of Stream Ciphers

Jury:

Prof. Dr. ir. Hendrik Van Brussel, voorzitter Prof. Dr. ir. Bart Preneel, promotor Prof. Dr. Lars R. Knudsen (TUD) Prof. Dr. ir. Vincent Rijmen Prof. Dr. ir. Marc Van Barel Prof. Dr. ir. Joos Vandewalle Proefschrift voorgedragen tot het behalen van het doctoraat in de ingenieurswetenschappen door

Hongjun Wu 伍 宏 军

U.D.C. 681.3*D46

July 2008

© Katholieke Universiteit Leuven – Faculteit Ingenieurswetenschappen Arenbergkasteel, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2008/7515/74

ISBN 978-90-5682-000-8

Acknowledgements

I would like to express my gratitude to Prof. Bart Preneel for providing me the opportunity to study at the Katholieke Universiteit Leuven and for being the promotor of this thesis.

I am very grateful to Prof. Lars R. Knudsen, Prof. Vincent Rijmen, Prof. Marc Van Barel and Prof. Joos Vandewalle for serving on the jury and to Prof. Hendrik Van Brussel for chairing the jury.

I am also thankful to all the COSIC members. I would like to thank Alex, Christophe, Christopher, Dawu, Elena, Emilia, Gautham, Joe, Jongsung, Jongsu, Junfeng, Kare, Li, Mina, Orr, Ozgul, Panagiotis, Sebastiaan, Souradyuti, Toga and Zhiguo for interesting discussions on various topics, and thank Pela for the help with administrative matters.

Special thanks go to Dr. Feng Bao and Prof. Robert H. Deng for being my supervisors for about eight years when I was a student at the National University of Singapore and when I was working at the Institute for Infocomm Research. I would like to thank my colleagues at the Institute for Infocomm Research for their friendship and helpful discussions. And I would like to thank Dr. Shaohua Tan for introducing me to cryptology.

Finally, many thanks to my girl friend Liming for her understanding and support during the last three years.

Hongjun Leuven, July 2008

iii

iv

Abstract

This thesis presents some novel results on the cryptanalysis and design of stream ciphers. The first part of the thesis introduces various stream ciphers design and cryptanalysis techniques.

The second part of the thesis gives the cryptanalysis of seven stream ciphers. The properties of addition are exploited in the cryptanalysis of two stream ciphers: the differential-linear cryptanalysis against Phelix and the fast correlation attack on ABC v2. Resynchronization attacks are applied against several stream ciphers – DECIM, WG, LEX, Py and Pypy. Various cryptanalytic approaches (linear, differential and slide attacks) are used in these attacks.

The third part of the thesis is on the design of stream ciphers. We demonstrate that strong and secure stream ciphers can be designed using nonlinear state updating function and nonlinear output function. The design of stream ciphers HC-256 and HC-128 are presented.

 \mathbf{V}

vi

Contents

| 1 | Intr | Introduction 1 | |
|---|------|--|--|
| | 1.1 | Symmetric Key Encryption | |
| | | 1.1.1 The one-time pad and stream ciphers | |
| | | 1.1.2 Block ciphers and stream ciphers | |
| | 1.2 | Stream Cipher Design 3 | |
| | | 1.2.1 Rotor machines | |
| | | 1.2.2 LFSR based stream ciphers $\ldots \ldots \ldots \ldots \ldots \ldots \ldots 5$ | |
| | | 1.2.3 Stream ciphers based on clock-controlled LFSRs 5 | |
| | | 1.2.4 Stream ciphers with nonlinear state update function 6 | |
| | | 1.2.5 Impractical stream ciphers | |
| | 1.3 | Stream Cipher Cryptanalysis | |
| | | 1.3.1 General attacks | |
| | | 1.3.2 Attacks on LFSR based stream ciphers | |
| | | 1.3.3 Unknown attacks | |
| | 1.4 | IV in Stream Cipher | |
| | 1.5 | Achievements | |
| | 1.6 | Outline | |
| 2 | Exr | loiting Characteristics of Addition I – Differential-Linear At- | |
| - | tack | c on Phelix 17 | |
| | 2.1 | Introduction | |
| | 2.2 | The Stream Cipher Phelix | |
| | 2.3 | The Differential Propagation of Addition | |
| | 2.4 | A Basic Key Recovery Attack on Phelix | |
| | | 2.4.1 The bias in the differential distribution of keystream 21 | |
| | | 2.4.2 Recovering the key | |
| | 2.5 | Improving the Attack on Phelix | |
| | | 2.5.1 Recovering $Z_4^{(i)}$ | |
| | | 2.5.2 Recovering $X_{i+1,0}$ | |

vii

| | $2.6 \\ 2.7$ | How to strengthen Helix and Phelix | $\frac{31}{31}$ | | |
|----------|----------------|--|-----------------|--|--|
| 3 | Exp | Exploiting Characteristics of Addition II – Fast Correlation At- | | | |
| | tack | c on the Stream Cipher ABC v2 | 33 | | |
| | 3.1 | Introduction | 33 | | |
| | 3.2 | The Stream Cipher ABC v2 | 34 | | |
| | 3.3 | The Weak Keys of ABC v2 | 36 | | |
| | | 3.3.1 The large bias of carry bits | 36 | | |
| | 0.4 | 3.3.2 Identifying the weak keys | 37 | | |
| | 3.4 | Recovering the Internal State | 39 | | |
| | | 3.4.1 Recovering the initial value of the LFSR | 39 | | |
| | | 3.4.2 Recovering the components B and C | 40 | | |
| | | 3.4.3 The complexity of the attack | 41 | | |
| | 0.5 | 3.4.4 The attack on ABC v1 | 42 | | |
| | 3.5 | Conclusion | 42 | | |
| 4 | \mathbf{Res} | ynchronization Attack I – Linear Attack on DECIM | 45 | | |
| | 4.1 | Introduction | 45 | | |
| | 4.2 | Stream Cipher DECIM | 46 | | |
| | | 4.2.1 Keystream Generation | 46 | | |
| | | 4.2.2 Initialization | 48 | | |
| | 4.3 | Key Recovery Attack on DECIM | 48 | | |
| | | 4.3.1 The effects of the permutations π_1 and π_2 | 48 | | |
| | | 4.3.2 Recovering K_{21} | 49 | | |
| | | 4.3.3 Recovering $K_{22}K_{23}K_{30}$ | 50 | | |
| | | 4.3.4 Recovering $K_9K_{10}\ldots K_{19}$ | 50 | | |
| | | 4.3.5 Recovering $K_{32}K_{33}K_{46}$ | 51 | | |
| | 4.4 | Improving the Key Recovery Attack | 51 | | |
| | 4.5 | The Keystream of DECIM Is Heavily Biased | 51 | | |
| | | 4.5.1 The keystream is biased | 52 | | |
| | | 4.5.2 Broadcast attack | 53 | | |
| | 4.6 | Attacks on DECIM with 80-bit IV | 53 | | |
| | 4.7 | Conclusion | 54 | | |
| 5 | Res | vnchronization Attack II – Differential Attack on WG | 55 | | |
| 0 | 5.1 | Introduction | 55 | | |
| | 5.2 | Description of WG | 56 | | |
| | 5.3 | Differential Attacks on WG | 58 | | |
| | 0.0 | 5.3.1 Attack on WG with an 80-bit key and an 80-bit IV | 58 | | |
| | | 5.3.2 Attacks on WG with key and IV sizes larger than 80 bits | 61 | | |
| | | 5.3.3 Attacks on WG with 64-bit IV size | 61 | | |
| | | SIGIS TRUCKS OF THE WITH OF DID IV DIDE | 01 | | |

viii

| | 5.4 | Conclusion | 62 |
|---|-------------------|---|-----------------|
| 6 | Res 6.1 | ynchronization Attack III – Slide Attack on LEX Introduction | 65 65 |
| | 6.2 | Description of LEX | 65 |
| | 6.3 | Slide Attack on the Resynchronization of LEX | 66 |
| | 6.4 | Conclusion | 68 |
| 7 | Res | ynchronization Attack IV – Differential Attack on Py, Py6 | |
| | and | Руру | 69 |
| | 7.1 | Introduction | 69 |
| | 7.2 | The Specifications of Py and Pypy | 70 |
| | | 7.2.1 The key setup \ldots | 71 |
| | | 7.2.2 The IV setup | 71 |
| | | 7.2.3 The keystream generation | 73 |
| | 7.3 | Identical Keystreams | 73 |
| | | 7.3.1 IVs differing in two bytes | 74 |
| | | 7.3.2 IVs differing in three bytes | 76 |
| | | 7.3.3 Improving the attack | 77 |
| | 7.4 | Key Recovery Attack on Py and Pypy | 78 |
| | | 7.4.1 Recovering part of the array Y | 78 |
| | | 7.4.2 Recovering the key \ldots \ldots \ldots \ldots \ldots \ldots | 81 |
| | 7.5 | The Security of Py6 | 82 |
| | 7.6 | Conclusion | 83 |
| 8 | The | Stream Cipher HC-256 | 85 |
| | 8.1 | Introduction | 85 |
| | 8.2 | Stream Cipher HC-256 | 86 |
| | | 8.2.1 Operations, variables and functions | 86 |
| | | 8.2.2 Initialization process (key and IV setup) | 87 |
| | | 8.2.3 The keystream generation algorithm | 88 |
| | | 8.2.4 Encryption and decryption | 88 |
| | 8.3 | Security Analysis of HC-256 | 89 |
| | | 8.3.1 Period | 89 |
| | | 8.3.2 The security of the key | 89 |
| | | 8.3.3 Randomness of the keystream | 90 |
| | | 8.3.4 Security of the initialization process (key/IV setup) | 94 |
| | 8.4 | Implementation and Performance of HC-256 | 94 |
| | | 8.4.1 The optimized implementation of HC-256 | 95 |
| | | 8.4.2 Performance of HC-256 | 96 |
| | 8.5 | Conclusion | 97 |
| | | | |

ix

| 9 | The | e Stream Cipher HC-128 99 |) |
|--------------------|------|---|---|
| 9.1 Introduction | |) | |
| | 9.2 | Cipher Specifications |) |
| | | 9.2.1 Operations, variables and functions |) |
| | | 9.2.2 Initialization process (key and IV setup) | 1 |
| | | 9.2.3 The keystream generation algorithm | 1 |
| | | 9.2.4 Encryption and decryption | 2 |
| | 9.3 | Security Analysis of HC-128 | 2 |
| | | 9.3.1 Period | 2 |
| | | 9.3.2 Security of the secret key $\ldots \ldots \ldots$ | 3 |
| | | 9.3.3 Security of the initialization process (key/IV setup) 103 | 3 |
| | 9.4 | Randomness of the keystream | 3 |
| | 9.5 | Implementation and Performance of HC-128 | 5 |
| | | 9.5.1 The optimized implementation of HC-128 108 | 5 |
| | | 9.5.2 The performance of HC-128 | 3 |
| | 9.6 | Conclusion | 3 |
| 10 Conclusions 107 | | | |
| Α | The | e Number of IVs to Break DECIM 119 |) |
| в | Test | t Vectors of HC-256 and HC-128 123 | 3 |
| | B.1 | Test Vectors of HC-256 | 3 |
| | B.2 | Test Vectors of HC-128 | 1 |

х

List of Figures

| 2.1 | One block of Phelix | 19 |
|--------------|---------------------------------------|-----------------|
| 3.1 | Keystream generation of ABC v2 | 35 |
| 4.1 | Keystream Generation Diagram of DECIM | 47 |
| $5.1 \\ 5.2$ | Keystream generation diagram of WG | $\frac{56}{57}$ |
| $6.1 \\ 6.2$ | Initialization and stream generation | 66 66 |

xi

xii

List of Tables

| $2.1 \\ 2.2 \\ 2.3$ | The probability that $B_3^{(i+1),j} \oplus B_3^{\prime(i+1),j} = 0$ for $P_i \oplus P_i^{\prime} = 1$ The probability that $Y_4^{(i+1),j} \oplus Y_4^{\prime(i+1),j} = 0$ for $P_i \oplus P_i^{\prime} = 1$ The number of plaintext pairs for recovering $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$ | 21 25 29 |
|---------------------|---|----------------|
| 3.1 | The probability of $c_1 \oplus c_2 \oplus c_3 = 0$ (denote the probability as $\frac{1}{2} + \epsilon$) | 37 |
| 5.1 | The differential propagation in the key/IV setup of WG $\ . \ . \ .$. | 60 |
| 8.1 | The speed of the C implementation of HC-256 on a Pentium 4 $$ | 96 |
| A.1 A.2 | Number of IVs required to recover the key bits (64-bit IV) 1 Number of IVs required to recover the key bits (80-bit IV) 1 | 120 121 |

xiii

xiv

List of Symbols

| + | $x + y$ means $x + y \mod 2^{32}$, where $0 \le x < 2^{32}$ and $0 \le y < 2^{32}$ |
|----------|---|
| \oplus | bit-wise exclusive OR |
| | concatenation |
| \gg | right shift operator. $x \gg n$ means x being right shifted over n |
| | bit positions. |
| \ll | left shift operator. $x \ll n$ means x being left shifted over n bit positions. |
| >>>> | right rotation operator. $x >>> n$ means x being rotated to the right over n bit positions. |

 $\mathbf{X}\mathbf{V}$

xvi

List of Abbreviations

| AES | Advanced Encryption Standard |
|----------------|--|
| \mathcal{AI} | Algebraic immunity |
| BIMAG | Bistable Magnetic core |
| CBC | Cipher-block chaining mode |
| CFB | Cipher feedback mode |
| CMOS | Complementary metal-oxide-semiconductor |
| CTR | Counter mode |
| DES | Data Encryption Standard |
| ECB | Electronic Codebook |
| EMP | Electromagnetic pulse |
| GSM | Global System for Mobile communications |
| IV | Initialization Vector |
| LC | Linear Complexity |
| LFSR | Linear Feedback Shift Register |
| MAC | Message authentication code |
| NIST | National Institute of Standards and Technology (USA) |
| NLFSR | Nonlinear Feedback Shift Register |
| NSA | National Security Agency (USA) |
| OFB | Output feedback mode |
| PKI | Public Key Infrastructure |
| SSL | Secure Socket Layer |
| XOR | Exclusive OR |

xvii

xviii

Chapter 1

Introduction

1.1 Symmetric Key Encryption

The importance of encryption became critical after telegraph, especially radio telegraph, was invented. Long distance communication allows information being intercepted much easier than ever. To protect the confidentiality of information, encryption is widely used in military, intelligence and diplomatic services. The consequence is that cryptanalysis techniques improved significantly. During World War II, both the German Enigma cipher [53, 123] and the Japanese Purple cipher [52] were successfully broken by the Allies. The two weak ciphers contributed significantly to the failure of Germany and Japan in World War II.

Today more and more people are connected to the internet with huge amount of confidential information (emails, online transactions ...) being transmitted every day. Cryptography starts to play an important role in daily life. Modern cryptography is developed to protect information confidentiality, integrity and provide authentication. In modern cryptography, symmetric key cipher is essential in protecting information confidentiality. With a public key infrastructure (PKI) that can support key establishment protocol, two parties can share a secret key and carry out symmetric key encryption in a convenient way.

Symmetric key encryption is important for secret information transmission and storage. Two parties, the sender and receiver, share the same symmetric key cipher and the same secret key. The sender encrypts the message (plaintext) with the cipher and key to obtain the ciphertext. The ciphertext is transmitted (or stored) over an insecure channel. The receiver decrypts the ciphertext to retrieve the original message. An attacker or adversary may intercept the ciphertext. To ensure that no information is leaked to the attacker, strong cipher and strong key should be used for encryption.

1

1.1.1 The one-time pad and stream ciphers

The one-time pad, also called Vernam's cipher, was invented by Vernam in 1917. The bit-wise one-time pad is easy to illustrate. A one-time key is randomly generated and it is as long as the message. The key is XORed (addition modulo 2) with the plaintext for encryption, and the key is XORed with the ciphertext for decryption.

The one-time pad is the only encryption algorithm that is unconditionally secure – the cipher remains secure despite the computational power and knowledge of any adversary (quantum cryptography is also claimed to be "unconditionally secure", but such claim is based on the assumption that the physics knowledge today is perfect). The perfect secrecy of one-time pad was proved by Claude Shannon in his historic paper in 1949 [118].

Although the one-time pad is perfectly secure, it is inconvenient to use in many applications due to the constraints that the key is too long and each key can be used only once. A strong synchronous stream cipher is a good replacement for the one-time pad. A stream cipher can be used to generate many keystreams from the same (relatively short) key and different initialization vectors (IVs), then each keystream can be used to encrypt a message.

1.1.2 Block ciphers and stream ciphers

Symmetric key encryption algorithms include block ciphers and stream ciphers. Both block ciphers and stream ciphers are widely used today. A block cipher has a fixed message input length, called block size, and it can be viewed as an enormous and fixed (for each key) secret substitution table that transforms a block of plaintext bits into ciphertext. A stream cipher has a variable message input length, and it can be viewed as a small but changing secret substitution table that transforms plaintext bits at different positions with different substitution tables (the XOR operation between plaintext and keystream can be viewed as one-bit substitution determined by a keystream bit). There is some connection between block ciphers and stream ciphers. A block cipher in counter (CTR) mode [98] or output feedback (OFB) mode [98] is an inefficient synchronous stream cipher; and a block cipher in cipher feedback (CFB) mode [98] is an asynchronous stream cipher.

For some applications, a block cipher is more convenient to use than a stream cipher – block cipher in electronic codebook (ECB) mode [98] does not require IV; and the security of block cipher in cipher-block chaining (CBC) mode [98] is not that sensitive to the management of IV. A block cipher is also useful in building other cryptography primitives, such as hash functions and message authentication codes (MACs). A stream cipher has two advantages over a block cipher. One advantage is that to achieve the *same* security level, a stream cipher

may require much less computations than a block cipher because it is more difficult to attack the changing state in stream cipher. Another advantage is that with the precomputation of keystream, the encryption and decryption of stream cipher can be extremely fast.

The Data Encryption Standard (DES) [99] and Advanced Encryption Standard (AES) [101] are the two block cipher standards of the National Institute of Standards and Technology (NIST) of the United States of America. DES was selected as the encryption standard in 1976, and it has motivated the research on the design and cryptanalysis of block ciphers. AES was selected as standard in 2001. In the following, we give a brief introduction to the design and analysis of stream ciphers.

1.2 Stream Cipher Design

A stream cipher consists of a state update function and an output function. The state of a stream cipher is updated continuously during encryption so that bits at different positions in a message are encrypted with different states. The output function generates keystream bits from the state and performs encryption or decryption. If the initial state of a stream cipher is not the same as the key, key setup is required to generate the initial state from the key. If a key is used with different initialization vectors (IVs) to generate keystreams, key/IV setup (resynchronization) is required to generate the initial state from the key and IV.

Stream ciphers can be classified according to the state update. If the state is independent of the message, the cipher is called a synchronous stream cipher since it requires synchronization between the sender and receiver. If the state depends only on N previous ciphertext bits, the cipher is called asynchronous or self-synchronizing stream cipher. Some stream ciphers, such as Helix [48] and Phelix [126], are neither synchronous nor asynchronous since the state is affected by all the previous message bits.

The basic requirement on the state update is that the states should be generated with a sufficiently large period. There are many ways to design a state update function. The simplest way is to use counter, such as block cipher in counter mode and the stream cipher Salsa20 [19]. There are two problems with a counter. One problem is that the diffusion within a counter is too slow; another problem is that the more significant bits never affect the less significant bits, so the cipher with counter requires a lot of computations in order to achieve a high security level. A more sophisticated approach is to use linear feedback shift register (LFSR) with a primitive polynomial. To reduce the weakness associated with the linearity in LFSR, nonlinearity is introduced, such as clocking the LFSR irregularly. The more general approach to eliminate the weak linearity in an LFSR is to use a nonlinear state update function instead of LFSR. Many different and efficient stream ciphers have been designed with nonlinear state update functions in recent years. We expect that such approach will continue to dominate the future stream cipher designs.

In the following, we will illustrate the above state update techniques in detail. We start with the early stream ciphers, rotor machines, to see how a counter can be constructed in an electromechanical way.

1.2.1 Rotor machines

A cipher is practical only if its encryption and decryption can be computed in an efficient way. Any cipher used in history matches well with the computing facilities at that time. A stream cipher with state update function is difficult to compute conveniently and reliably with paper and pencil. With the application of electromechanics, the early stream ciphers, rotor machines cipher, were invented and became widely used from 1930s to 1960s. Rotor machines are convenient to use – encryption and decryption only requires an operator typing a message on the keyboard. Rotor machine is a brilliant cipher design approach in the electromechanics era.

Enigma is the best known rotor machine since it was used in the German military during World War II and was broken by the Allies. The rotors of Enigma are described below. Enigma consists of a few rotors, typically 3 or 4, driven by electricity. Each rotor represents a (secret) substitution table of the German characters; the concatenation of two rotors gives the composition of two substitutions. Clocking a rotor one step is equivalent to rotate the substitution table one position. For a common Enigma (the naval version is slightly different), the first rotor is rotated one step for every input character, and it triggers the second rotor to rotate one step during every round of the first rotor; the second rotor triggers the third rotor to rotate one step during every round of the second rotor. The initial rotor positions are set with a few characters (initialization vector). The state of Enigma keeps changing as the character in the message. Enigma is constructed with an additional reflector so that the encryption and decryption are identical.

The state update function of Enigma is a simple counter. For Enigma with three rotors, the changing state (the positions of rotors) can be viewed as a three digit number in base 26 (for English language), and this number is incremented each time a key is typed. The value of the counter determines the substitution tables to be used in the encryption and decryption. A counter is probably the simplest state update function that can be implemented easily and reliably with electrical rotors (or relays), so rotor machines emerged in the early 20th century and became popular for about four decades.

1.2.2 LFSR based stream ciphers

Electronic computers were invented in the 1940s. The advance in computing technology allowed cryptographers to design ciphers using more secure operations than counters. The LFSR based stream cipher started to emerge at that time. The algebraic structure of LFSR is very simple and it can be easily constructed using basic logic gates.

In the 1950s, the National Security Agency (NSA) of the United States of America started the design of the KW-26 stream cipher, as illustrated in the NSA brochure [102]. KW-26 was in service from 1960s to 1980s. It was constructed using over 800 Bistable Magnetic cores (BIMAGs) and about 50 vacuum-tube drivers. BIMAG is very reliable, and was used as memory in almost all the computers in the 1950s and early 1960s. BIMAGs and vacuum tubes are much less vulnerable to electromagnetic pulse (EMP) and radiation, so they are very attractive to military applications. It is thus not a surprise that BIMAG and vacuum tube were used in the construction of KW-26 even when silicon technology became popular in the 1970s. The specification of KW-26 has never been disclosed to the public and it is uncertain how many versions have been developed and used. After KW-26 ciphers were decomissioned by NSA in the mid-1980s, the ciphers were securely destroyed. According to the NSA brochure, KW-26 uses Fibonacci shift registers and binary logic combining elements; it is very likely that the term "Fibonacci shift registers" refers to linear feedback shift registers.

There are two types of LFSR based stream ciphers. One type is to use a long LFSR, and to use a filtering function to generate keystream bits; another type is to use several short LFSRs, and to use a combining function to generate keystream bits. The filtering or combining function has to involve complicated operations in order to hide the linear weakness in the regularly clocked LFSR(s), and it is related to the study of the properties of Boolean functions, such as resilience, nonlinearity and algebraic immunity.

1.2.3 Stream ciphers based on clock-controlled LFSRs

A regularly clocked LFSR is weak. The linearity in the LFSR allows to combine information that is leaked at faraway positions in order to recover the secret state of the LFSR. To reduce the weakness, nonlinearity can be introduced to an LFSR by clocking the LFSR irregularly.

In the stop-and-go generator [24], two LFSRs are cascaded – one bit from LFSR1 is used to control the clocking of LFSR2 following the stop-and-go rule, and one bit from LFSR2 is given as the output of the generator. The weakness of the stop-and-go generator is that when LFSR2 is not clocked, the bit from LFSR2 is still given as the output of the generator. This weakness is eliminated perfectly in the alternating step generator [64]. In the alternating step generator,

one bit from LFSR1 is used to control the clocking of LFSR2 and LFSR3 so that one of LFSR2 and LFSR3 is clocked at each step. Two bits, one from LFSR2 and another from LFSR3, are XORed to give the output of the generator. However, both the stop-and-go generator and alternating step generator are vulnerable to the divide-and-conquer attack since guessing the initial state of LFSR1 is almost equivalent to guessing the complete initial state of the generator.

The above weaknesses in the stop-and-go generator and alternating step generator are eliminated in the elegant stream cipher A5/1 [59], which is used in GSM. A5/1 consists of three LFSRs, each LFSR provides a control bit, and an LFSR is clocked if its control bit agrees with the majority of those three control bits, thus at least two LFSRs are clocked at each step. Each LFSR provides one bit to be XORed together to generate the keystream bit. The three LFSRs in A5/1 affect one another, so A5/1 is not that vulnerable to the divide and conquer attack, and the size of each LFSR can be small. A5/1 can be implemented very efficiently in hardware due to its small state size and simple operation. Several attacks have been developed against A5/1 by exploiting its small 64-bit state size [59, 22, 9]. However, considering that the key size of A5/1 is limited to only 54 bits, A5/1 almost achieves the 54-bit security level required in GSM.

The above clock controlled generators were developed in the 1980s. The shrinking generator [34] and self-shrinking generator [90], being proposed in the 1990s, are two special clock controlled generators. There are two regularly clocked LFSRs in the shrinking generator. At each step, one bit from LFSR1 is used to determine whether a bit from LFSR2 is given as keystream bit or discarded. There is only one regularly clocked LFSR in the self-shrinking generator. At the end of every two steps, two adjacent bits in the LFSR are used – one bit is used to determine whether another bit will be given as keystream bit or discarded. There are so far no efficient attacks against the shrinking generator and self-shrinking generator. However, it is inconvenient to use the shrinking generators in practice since they do not generate keystream at a constant rate.

From the clock controlled generators, we see that nonlinearity in the state update function can result in a simple, secure and efficient design. It is natural to look for other efficient nonlinear state update functions in the stream cipher design.

1.2.4 Stream ciphers with nonlinear state update function

Computing devices are very powerful these days. We can design many different nonlinear state update functions – irregularly clocked LFSRs, bit-wise nonlinear feedback shift registers, lookup table based nonlinear update functions ... With nonlinear state update functions, very secure and efficient stream ciphers can be designed.

Bit-wise nonlinear feedback shift registers

The bit-wise nonlinear feedback shift registers (NLFSR) are used in several recent stream ciphers, such as Achterbahn [54, 55], Grain [66, 67] and Trivium [44]. Achterbahn consists of a number of short NLFSRs, but it does not benefit much from the use of NLFSRs. Even though complicated combining functions are used in Achterbahn, the ciphers were still broken [76, 68, 69, 111]. Grain consists of one LFSR and one NLFSR. The improperly designed NLFSR and filtering function in Grain v0 results in the state of LFSR being recovered with a fast correlation attack [15]. The flaw was fixed in the improved Grain version [67]. Trivium uses a long NLFSR (or three NLFSRs with interconnections between them), and is very strong against the distinguishing attack.

We expect that extremely hardware efficient stream ciphers can be designed by using bit-wise NLFSRs and nonlinear output functions. However, as the number of operations being significantly reduced in the hardware efficient cipher, it requires more security analysis of the design.

Word-wise nonlinear update functions

RC4 [114] is used extensively in applications such as Secure Socket Layer (SSL). It is the first lookup table based stream cipher. Table lookups are used extensively in the state update and keystream generation in RC4. The table in RC4 is secret and changing. Benefiting from this high nonlinearity, the structure of RC4 is extremely simple, and RC4 is very strong. However, the initial state of RC4 is not properly generated from the key and IV, thus it suffers from the resynchronization attack [50, 84] and broadcast attack [86] in practice. In general, it is extremely difficult to recover the secret key of a stream cipher with large, secret and changing lookup table by analyzing the keystream generation process. The recent designs with secret and changing lookup table include HC-128 [135], HC-256 [128], Py [25] and Pypy [26]. However, Py and Pypy suffer from resynchronization attacks [134].

Rabbit [30] is a stream cipher with update function heavily depending on integer multiplications and rotations. This approach also provides high nonlinearity in the cipher design.

More on nonlinear update functions

When a nonlinear update function is used in a stream cipher, the most important question is what the period of the keystream is. If there is no algebraic structure in the nonlinear function, then it is almost impossible to predict the exact period of the keystream. If algebraic structure is introduced into the nonlinear function, such as a T-function [79, 80], then it is possible to compute the exact period of the keystream. However, we prefer removing any algebraic structure from the stream

cipher design to reduce the threat that the algebraic structure may be exploited in an attack. In general, if the state size of a stream cipher is not too small (at least twice the keysize), and the state is updated in an *invertible* and nonlinear way (with sufficient confusion and diffusion), then the probability is very high that the resulting stream cipher generates keystream with a very large period since the average period of an invertible random function with *n*-bit state is 2^{n-1} [49]. We should note that if the nonlinear update function is noninvertible, then the average period of the keystream would be significantly reduced, such as the short period of A5/1.

We should also note that using nonlinear state update function in a stream cipher does not automatically guarantee the security of the cipher. Confusion and diffusion are very important in stream cipher design, although they have been strictly applied in block cipher design. And intensive reviews are necessary for any stream cipher design.

1.2.5 Impractical stream ciphers

Some stream ciphers, such as QUAD [16], are designed with public key cryptosystem techniques. This approach normally gives stream cipher more than 100 times less efficient than a common stream cipher and thus has almost no practical value. Basically such approach is a bit ridiculous: in order to achieve "provable security", the designers have to significantly weaken the cipher by giving "public key" information to the adversary. Obviously, keeping such "public key" information secret in a symmetric key cipher should significantly improve the security (or significantly enhance the performance) – just imagine how secure (or how efficient) the properly padded RSA [115] would be if both the public and private keys are kept secret.

1.3 Stream Cipher Cryptanalysis

Cryptanalysis plays an essential role in the design of ciphers. A good cipher should be designed by taking into account all the known cryptanalysis techniques and the designer's insight into unknown attacks. For example, DES would not have been designed in the same way if the differential attack [27] had not been invented at that time, and AES would not give adequate security margin if the square attack [42] had not been developed at that time. In the following, we illustrate some general attacks on stream ciphers, followed by the dedicated attacks on LFSR based stream ciphers. The countermeasure against these attacks will be discussed.

1.3.1 General attacks

There are a number of general attacks against stream ciphers. These include the brute force attack, the time-memory-data tradeoff attack, the divide and conquer attack (including correlation attack), the resynchronization attack and distinguishing attack.

Brute force attack

A brute force attack (exhaustive key search) is the most basic attack against any cipher. The key space of a cipher should be sufficiently large to thwart brute force attack. The adequate key size is closely related to the security requirement and the advance in computing technology.

Computing can be simply considered as the flow of information and the triggering of events during the flow. Thus computing power is eventually limited by the information flow distance and transmission speed. Within a conventional computer, the information transmission speed is limited by the speed of light, and the information transmission distance can be reduced through the aggressive shrinking of transistors. The shrinking of transistors has so far been predicted well by Moore's law which states that the number of transistors that can be placed on an integrated circuit doubles every 18 months. However, there will be an end to Moore's law. In 2007, Intel showed the first test chips using the 32 nanometer node CMOS fabrication technology (half-pitch spacing of metal lines is 32 nanometers). The 16 nanometer technology is expected to be available in 2018. Note that the diameter of an atom is about 0.1 to 0.5 nanometer, hence we can expect that Moore's law would no longer be applicable after three or four decades. It implies that stream cipher with 128-bit key size may be secure enough against brute force attack within the first half of the 21st century.

However, there is uncertainty in predicting the progress of computing. There will be alternative ways to perform computing rather than using the current silicon technology (microelectronics). The future researches on nanoelectronics will very likely lead to much more powerful computing devices. Quantum computing is an emerging and very different computing approach. Based on entanglement and superposition, an *n*-qubit quantum register can contain up to 2^n states simultaneously, and an operation performed on the quantum register is equivalent to 2^n parallel operations being performed on a conventional computer. It is the reason that quantum computers can be much more powerful than conventional computers for solving a general problem. The reason is that although a quantum register contains 2^n states, it can only be measured to be one of 2^n states, so it is extremely difficult to exploit the quantum computing power directly. Thus quantum algorithms are needed so that some special problems

can be solved efficiently on quantum computers. There are so far two quantum algorithms being developed that are important for cryptanalysis – Shor's algorithm [119] and Grover's algorithm [63]. Shor's algorithm was invented by Peter Shor in 1994. It is the first quantum algorithm related to cryptology and it has motivated the research on quantum computer. Shor's algorithm is a polynomialtime approach to solve the integer factorization and discrete logarithm problems. Thus it is easy to break most of the widely used public key cryptosystems on a quantum computer. Grover's algorithm searches an unsorted database with N entries in $O(\sqrt{N})$ time with $O(\log N)$ storage space. Grover's algorithm can be applied to find an *n*-bit key with about $2^{n/2}$ operations, thus it affects the security of symmetric key ciphers. However, it is unlikely that quantum computer can be applied to break stream cipher faster than conventional computer in the near future (say, half a century), since currently quantum computers are still in their infancy, and there are tremendous difficulties in building quantum computers. The threat of Grover's algorithm on stream ciphers can be simply eliminated by doubling the key size.

Time-memory-data tradeoff attack

If the state space of a stream cipher is small, the collision of the states can be exploited to recover the secret state. The attack is performed as follows. From many different states, the corresponding keystream fragments are precomputed, sorted and stored. From a given long keystream, one can obtain many keystream fragments, and compare the keystream fragments with the precomputed keystream fragments to look for collision. Once a collision is found, the secret state of the cipher is recovered. The above attack was developed independently by Babbage [7] and Golić [59]. Note that if part of the secret state of a stream cipher is never updated by the state update function, then collision does not exist for that fixed state component, so brute force attack is needed to deal with the fixed secret state of attack increases.

In the above attack, there is tradeoff between the amount of precomputations and the length of keystream. The length of the known keystream can be reduced by increasing the amount of precomputations. But the memory required to store the precomputed keystream fragments and the corresponding states increases proportionally to the amount of precomputations. To reduce the memory requirement in the attack, in year 2000, Biryukov and Shamir developed the timememory-data tradeoff attack [21] by using the idea of Hellman's time-memory tradeoff attack on block cipher [71].

To resist the time-memory-data tradeoff attack, it requires that the state size is at least twice of the key size.

Divide and conquer attack

Divide and conquer is a natural and powerful approach to solve a complicated problem. We can find the use of divide and conquer in many attacks, such as differential cryptanalysis [27] and linear cryptanalysis [88] against block ciphers. In order to thwart the divide and conquer attack, the repeated use of confusion and diffusion is necessary in the design of a symmetric key cipher, as pointed out by Shannon [118].

Here we consider only the divide and conquer attacks against stream ciphers with improperly updated state – some component in the state is updated without being affected by other components. Many stream ciphers have this potential weakness, such as stream ciphers consisting of several independently updated linear (or nonlinear) feedback shift registers. T-function based stream cipher is a special example [79, 80]. In T-functions, the more significant bits never affect the less significant bits. The stream cipher Edon80 [56] does not use T-function. However, the overall structure of Edon80 is very similar to a T-function due to the absence of feedback. This weak structure was exploited to break Edon80 [70].

The correlation attack, developed by Siegenthaler in 1984 [120], is one type of divide-and-conquer attacks. It was originally developed against stream ciphers consisting of several LFSRs. There is always correlation between an LFSR and the keystream bits. If the LFSR length is short and the correlation is large, an attacker can search through all the possible initial states of LFSR, and find the correct initial state by checking the correlation between each guessed LFSR sequence and the keystream. In general, the correlation attack can be applied to any stream cipher in which the state consists of small and independently updated components.

To resist the above divide and conquer attack on stream cipher, a simple and efficient approach is to implement fast diffusion between all the state bits, i.e., each state bit being updated by all the other state bits frequently.

Resynchronization attack

Resynchronization attack targets the key/IV setup of stream ciphers. The security of the key/IV setup of stream cipher is critical. The key/IV setup with only linear operations is extremely risky because there is even no confusion in the setup. This weakness was exploited in several attacks [41, 31, 61]. Furthermore, a key/IV setup with insufficient nonlinear operations is still insecure, as illustrated below.

The reuse of key with many different IVs allows the key/IV setup being attacked similar (but not identical) to attacks on block ciphers. So we should take into account all the block cipher attacks when we design the key/IV setup. Differential cryptanalysis and linear cryptanalysis are two powerful attacks against block ciphers. In the two attacks, the correlation between plaintext and ciphertext is exploited to recover the secret key – differential cryptanalysis exploits the biased distribution of the difference of ciphertext pairs corresponding to a particular plaintext difference pattern; and linear cryptanalysis exploits the biased distribution of the parity of some plaintext and ciphertext bits. These two attacks are also useful in attacking the key/IV setup of stream cipher. We have applied differential attacks to break the key/IV setup of WG [131], Py and Pypy [132], and applied linear attacks to break the key/IV setup of DECIM [130].

To resist the resynchronization attack, there should be sufficient confusion and diffusion in the key/IV setup, similar to (but more difficult than) block cipher design.

Distinguishing attack

Randomness of the keystream is an important requirement for stream ciphers. A bias in the keystream can be applied to distinguish keystream from a random sequence. A large bias in keystream can be exploited to perform broadcast attack to recover the message when the same message is protected by different keystreams. The randomness requirement on keystream so far remains ambiguous despite many years of research on stream cipher. For pure academic research, a distinguishing attack with complexity less than brute force is acceptable. However, we believe that an attack that recovers less than one bit of information on the message from every 2^{64} ciphertext bits by exploiting the randomness of the keystream has negligible effect on applications.

We have mentioned earlier that RC4 is vulnerable to the broadcast attack. A recent stream cipher, DECIM [11], also succumbs to the broadcast attack due to the heavily biased keystream [130]. An interesting observation is that Enigma is extremely vulnerable to the broadcast attack. The encryption of Enigma is heavily biased – each character is never encrypted to itself. It is thus extremely risky to send a message to multiple receivers (say, 20) with different keys or initialization vectors; otherwise the original message can be easily recovered from those different ciphertexts without knowing the key. The strong bias had also been exploited extensively in recovering the secret state of Enigma by the Allies. To reduce the bias, each character should be encrypted repeatedly with a few Enigma ciphers.

1.3.2 Attacks on LFSR based stream ciphers

The most significant advancements in stream cipher cryptanalysis so far are the attacks on LFSR based stream ciphers. The linearity in LFSR provides a rich resource for developing several amazing attacks.

Berlekamp-Massey algorithm

The linear complexity (LC) of a sequence is the length of the shortest LFSR which can generate that sequence. The Berlekamp-Massey algorithm [18, 87] is an efficient algorithm for determining the linear complexity. The complexity of the algorithm is $O(n^2)$, where n is the length of the given sequence. The complexity is reduced to $O(n(\log n)^2 \log \log n)$ by using the Blahut algorithm [29]. The Blackburn algorithm [28] achieves the same complexity as the Blahut algorithm, but the Blackburn algorithm is easier to implement.

For stream cipher analysis, Berlekamp-Massey algorithm is useful in measuring the randomness of the keystream, and is powerful in generating the rest of the keystream from the known keystream bits if the linear complexity of the keystream is not large enough. Here, we can view Berlekamp-Massey algorithm as an efficient way of solving the regularly structured linear equations; otherwise, the complexity of the attacks is $O(n^{2.807})$ by using Strassen's algorithm [122] to solve these linear equations.

The linear complexity of the keystream of a stream cipher with a proper nonlinear update function may be quite close to (half of) the period of the keystream. The linear complexity of keystream of stream cipher based on regularly clocked LFSR is related to the nonlinear order of the filtering or combining function (Sect. 6.3 in [91]).

Fast correlation attack

For LFSR based stream ciphers, there is always a correlation between the LFSR bits and the keystream bits. Due to the linear nature of LFSR, the fast correlation attack can exploit the correlation at many bit positions to recover the LFSR. Let us consider the sequence from an LFSR in a stream cipher. According to the feedback polynomial and its multiples, each bit in this secret sequence is linearly related to other bits in the sequence through many linear relations. We apply these linear relations to keystream to see how many linear relations are satisfied for each keystream bit. Note that every bit in the keystream is correlated to a bit in the LFSR sequence. If a keystream bit satisfies most of the linear relations, then the chance is high that the value of the keystream bit is equal to the value of the bit in the LFSR sequence. Thus the correlation between the keystream bits and the bits in the LFSR sequence increases, and eventually the LFSR initial state could be recovered if the original correlation is large.

The fast correlation attack was invented by Meier and Staffelbach [89] in 1989. The attack was improved later [73, 74, 75, 93, 33, 32]. To resist the fast correlation attacks, a filtering or combining function with very small correlation between the input and output bits should be used in the design of LFSR based stream ciphers.

Algebraic attack and fast algebraic attack

For stream ciphers based on regularly clocked LFSR, the LFSR's initial state and keystream bits are connected through the nonlinear filtering or combining function. If the algebraic degree of the nonlinear function is low, the number of monomials in the nonlinear equations would be small, and the initial state of LFSR can be recovered by solving those overdefined nonlinear equations through the basic linearization technique by replacing each monomial with a new variable.

A filtering or combining function with high algebraic degree can resist the above attack. However, in 2003, Courtois and Meier made an important observation that if a polynomial is multiplied to both sides of a nonlinear Boolean equation, the algebraic degree of the resulting equation may be significantly reduced. Based on this observation, the algebraic attack on the LFSR based stream ciphers was developed [40]. The concept of algebraic immunity (\mathcal{AI}) of a boolean function was introduced to measure the resistance against the algebraic attack. But considering only the algebraic immunity in the design is insufficient. The fast algebraic attack [39] was developed to improve the algebraic attack by using consecutive keystream bits. For some Boolean function, the fast algebraic attack can reduce the degree further by exploiting the fact that Berlekamp-Massey algorithm is more efficient than Strassen's algorithm in solving the regularly structured linear equations. The fast algebraic attack was improved subsequently [1, 65].

When designing a stream cipher using regularly clocked LFSR, a designer has to evaluate the resistance of the Boolean function against the algebraic attack and fast algebraic attack. For a Boolean function involving too many input bits, the complexity of such an evaluation is high. The lowest complexity achieved so far is given in [2].

1.3.3 Unknown attacks

There is always uncertainty in the advance of cryptanalysis. It seems that if a cipher is simply over-optimized against the known attacks, then the chance is high that the cipher is vulnerable to some future attacks. However, for academic research, the performance of a cipher is critical, so the reality is that many ciphers have to be over-optimized for striking performance, especially for hardware performance. Anyway, it is not that bad since new attacks may be developed by analyzing the over-optimized ciphers.

Kerckhoffs' principle and the secrecy of ciphers

Kerckhoffs' principle states that a cipher should be secure even if the cipher specification, except the key, is public knowledge. Kerckhoffs' principle is a fundamental guideline in cipher design.

However, Kerckhoffs' principle does not mean that a cipher should be made public. Due to the uncertainty in cryptanalysis advancement, and the unequal cryptanalysis knowledge between the cipher designers and the adversary, the designers can at most assure that the cipher is secure to the designers at that moment, but they cannot assure that the cipher is secure against the adversary at that moment and in the future. Thus keeping a cipher secret is still necessary for top secret applications.

Here is a bit more on the unequal cryptanalysis knowledge. Enigma was broken in the early 1940s. However, the failure of Enigma was not disclosed until the 1970s. Differential cryptanalysis became publicly known in 1990. But it was known to NSA before DES was published in 1975. Clearly the statement that "a cipher should be secure" according to Kerckhoffs' principle is not that easy to achieve in practice.

1.4 IV in Stream Cipher

The initialization vector (IV) is very important in synchronous stream ciphers. It is disastrous if the same key is used with two identical IVs.

For a general purpose stream cipher, the IV size should be sufficiently large. The IV may be generated randomly, or be generated from a counter (the counter based IV is inconvenient to use in some applications), so we cannot simply assume that IV can only be generated from a counter. The IV size should be sufficient large to prevent the collision of IVs if IVs are generated randomly. Kohno pointed out in 2004 that the 64-bit IV being used in the AES-CTR in the compression software WinZip 9.0 [127] is not large enough to provide adequate security due to the collision of the IVs [83]. However, in the Ecrypt stream cipher project, eSTREAM [46], the minimum IV requirements are 32-bit IV for 80-bit key cipher, and 64-bit IV for 128-bit key cipher. Fortunately, many cipher submissions support 80-bit IV for 80-bit key or 128-bit IV for 128-bit key.

When a stream cipher is used in an application, the system designer should take special care of the IV update. We pointed out in 2005 that there is a serious security flaw in Microsoft Office XP [92] due to the poor IV management [129]. In Word and Excel 2002, RC4 is used for encryption, and MD5 is used to provide strong key/IV setup for RC4. The cipher itself is very strong. However, after an encrypted document gets edited, the IV does not change! It is disastrous since an IV is used for different version of a document. An adversary can recover a lot of information without knowing the key.

Since the security of stream cipher is very sensitive to the IV management, we recommend the use of a block cipher in CBC mode for the applications in which software or hardware performance is not critical, especially if the system designer is not familiar with stream cipher.

1.5 Achievements

eSTREAM [45] is the ECRYPT (European Network of Excellence in Cryptology) stream cipher project running from 2004 to 2008. From 34 stream cipher submissions, four software ciphers (HC-128, Rabbit, Salsa20/12 and SOSEMANUK [13]) and four hardware ciphers (F-FCSR-H v2 [17], Grain v1, MICKEY v2 [8] and Trivium) were chosen for the final portfolio. HC-128 is the fastest software stream cipher; Grain v1 and Trivium are very efficient in hardware.

This thesis includes attacks on seven eSTREAM candidates (Phelix, ABC v2, DECIM, WG, LEX, Py and Pypy) and one eSTREAM submission (HC-128 and HC-256). The attacks on six candidates recover the secret keys with low complexity, and can be easily carried out on a personal computer. HC-128 was chosen for the eSTREAM portfolio, and HC-256 is its 256-bit companion version.

- 1. Differential-linear attack on stream cipher Phelix [133]
- 2. Fast correlation attack on stream cipher ABC v2 [132]
- 3. Linear attack on the IV setup of stream cipher DECIM [130]
- 4. Differential attack on the IV setup of stream cipher WG [131]
- 5. Slide attack on the IV setup of stream cipher LEX [131]
- 6. Differential attack on the IV setup of stream ciphers Py and Pypy [134]
- 7. The stream ciphers HC-256 [128] and HC-128 [135]

1.6 Outline

The first part (Chapter 1) of the thesis covers various stream ciphers design and cryptanalysis techniques. We focus on our own views rather than the details of those techniques.

The second part of the thesis is on the cryptanalysis of stream ciphers. Chapter 2 and Chapter 3 exploit the properties of addition in the cryptanalysis of stream ciphers: the differential-linear cryptanalysis against Phelix and the fast correlation attack on ABC v2. Resynchronization attacks are applied against several stream ciphers – DECIM (Chapter 4), WG (Chapter 5), LEX (Chapter 6), Py and Pypy (Chapter 7). Various cryptanalysis approaches (linear, differential and slide attacks) are used in these attacks.

The third part of the thesis is on the design of stream ciphers. We demonstrate that strong and secure stream ciphers can be designed using nonlinear state updating function and nonlinear output function. Chapter 8 is on design of stream cipher HC-256. Chapter 9 is on the design of stream cipher HC-128.
Chapter 2

Exploiting Characteristics of Addition I Differential-Linear Attack on Phelix

Abstract. The previous key recovery attacks against Helix obtain the key with about 2^{88} operations using chosen nonces (reusing nonce) and about 1000 adaptively chosen plaintext words (or $2^{35.6}$ chosen plaintext words). The stream cipher Phelix is the strengthened version of Helix. In this chapter we apply differential-linear cryptanalysis to recover the key of Phelix. With 2^{34} chosen nonces and 2^{37} chosen plaintext words, the key of Phelix can be recovered with about $2^{41.5}$ operations.

2.1 Introduction

Phelix [126] is a fast stream cipher with embedded authentication mechanism. It is one of the focus ciphers (both software and hardware) of the ECRYPT eSTREAM project. Phelix is the strengthened version of the stream cipher Helix [48].

Muller has applied differential cryptanalysis to Helix [96]. It was shown that the key of Helix can be recovered faster than by brute force if the attacker can force the initialization vectors to be used more than once. The attack requires about 2^{12} adaptively chosen plaintext words and 2^{88} operations. Paul and Preneel reduced the number of adaptively chosen plaintext words by a factor of at least 3 [108]. Later Paul and Preneel showed that $2^{35.6}$ chosen plaintext words can be used instead of the adaptively chosen plaintext [107]. All these key recovery attacks against Helix require about 2^{88} computations.

To strengthen Helix, Phelix was designed and submitted to the ECRYPT

17

eSTREAM project. The output function of Helix has been changed so that a larger plaintext diffusion can be achieved in Phelix. The Phelix designers claimed that Phelix is able to resist the differential key recover attack even if the nonce is reused: "We claim, however, that even in such a case (referring to nonce reuse) it remains infeasible to recover the key" [126].

In this chapter, we apply differential-linear cryptanalysis to Phelix assuming nonce reuse (this corresponds to a chosen nonce attack). We show that the key of Phelix can be recovered with a low complexity: 2^{37} chosen plaintext words and $2^{41.5}$ operations.

This chapter is organized as follows. In Sect. 2.2, we illustrate the operations of Phelix. Section 2.3 analyzes how the addend bits affect the differential distribution. Section 2.4 describes a basic differential key recovery attack on Phelix. The improved attack is given in Sect. 2.5. We discuss how to strengthen Phelix in Sect. 2.6. Section 2.7 concludes this chapter.

2.2 The Stream Cipher Phelix

In this section, we only consider the encryption algorithm of Phelix. The full description of Phelix is given in [126]. The key size and nonce size of Phelix are 256 bits and 128 bits, respectively. The designers claim that there is no attack against Phelix with less than 2^{128} operations.

Phelix updates fives 32-bit words: Z_0 , Z_1 , Z_2 , Z_3 and Z_4 . At the *i*th step, two secret 32-bit words $X_{i,0}$, $X_{i,1}$ and one 32-bit plaintext word P_i are applied to update the internal states. One 32-bit keystream word S_i is generated and is used to encrypt the plaintext P_i . Note that the plaintext is used to update the internal state so that the authentication can be performed. The word $X_{i,0}$ is related to the key, and the word $X_{i,1}$ is related to the key and nonce in a very simple way. Recovering any $X_{i,0}$ and $X_{i,1}$ implies recovering part of the key. One step of Phelix is given in Fig. 2.1 [126].

2.3 The Differential Propagation of Addition

In this section, we study how the addend bits affect the differential propagation. The importance of this study is that it shows that the values of the addend bits can be determined by observing the differential distribution of the sum.

Theorem 2.1. Denote ϕ_i as the *i*th least significant bit of ϕ . Suppose two positive *m*-bit integers ϕ and ϕ' differ only at the *n*th least significant bit position $(\phi \oplus \phi' = 2^n)$. Let β be an *m*-bit random integer (*m* is much larger than *n*). Let $\psi = \phi + \beta$ and $\psi' = \phi' + \beta$. For $\beta_n = 0$, denote the probability that $\psi_{n+i} = \psi'_{n+i}$



Figure 2.1: One block of Phelix

as $p_{n+i,0}$. For $\beta_n = 1$, denote the probability that $\psi_{n+i} = \psi'_{n+i}$ as $p_{n+i,1}$. Then the difference $\Delta p_{n+i} = p_{n+i,0} - p_{n+i,1} = 2^{-n-i+1}$ (i > 0).

Theorem 2.1 can be proved easily if we consider the bias in the carry bits. We omit the proof here. In Theorem 2.1, the bias of the differential distribution decreases quickly as the value of n increases. We need another differential property that produces difference with a large bias even for large n. Before introducing that property, we give the following lemma.

Lemma 2.1. Denote u and v as two random and independent n-bit integers. Let $c_n = (u + v) \gg n$, where c_n denotes the carry bit at the nth least significant bit position. Denote the most significant bit of u as u_{n-1} . Then $\Pr(c_n \oplus u_{n-1} = 0) = \frac{3}{4}$.

Proof. $c_n = (u_{n-1} \cdot v_{n-1}) \oplus ((u_{n-1} \oplus v_{n-1}) \cdot c_{n-1})$. If $c_{n-1} = 0$, then $c_n \oplus u_{n-1} = u_{n-1} \cdot \overline{v}_{n-1}$, where \overline{v}_{n-1} denotes the inverse of v_{n-1} . If $c_{n-1} = 1$, then $c_n \oplus u_{n-1} = \overline{u}_{n-1} \cdot v_{n-1}$. Thus $\Pr(c_n \oplus u_{n-1} = 0) = \frac{3}{4}$.

The large bias of the differential distribution for large n is given below.

Theorem 2.2. Denote ϕ_i as the *i*th least significant bit of ϕ . Suppose two positive *m*-bit integers ϕ and ϕ' differ only at the *n*th least significant bit position $(\phi \oplus \phi' = 2^n)$. Let β be an *m*-bit random integer (*m* is much larger than *n*). Let $\psi = \phi + \beta$ and $\psi' = \phi' + \beta$. For $\beta_n \oplus \beta_{n-1} = 0$, denote the probability that $\psi_{n+i} = \psi'_{n+i}$ as $\bar{p}_{n+i,0}$. For $\beta_n \oplus \beta_{n-1} = 1$, denote the probability that $\psi_{n+i} = \psi'_{n+i}$ as $\bar{p}_{n+i,1}$. Then the difference $\Delta \bar{p}_{n+i} = \bar{p}_{n+i,0} - \bar{p}_{n+i,1} = 2^{-i}$ (i > 0).

Proof. Denote the carry bit at the *i*th least significant bit position in $\psi = \phi + \beta$ as c_i , and that in $\psi' = \phi' + \beta$ as c'_i . Note that $c'_n = c_n$, thus $c'_n \oplus \beta_n = c_n \oplus \beta_n$. When $c'_n \oplus \beta_n = c_n \oplus \beta_n = 0$, we know that $\psi \oplus \psi' = 2^n$ with probability 1, i.e., $\psi_{n+i} = \psi'_{n+i}$ with probability 1 for i > 0. When $c'_n \oplus \beta_n = c_n \oplus \beta_n = 1$, by induction we obtain that $\psi_{n+i} = \psi'_{n+i}$ with probability $1 - 2^{-i+1}$ for i > 0. According to Lemma 2.1, we know that $c_n \oplus \beta_{n-1} = 0$ with probability $\frac{3}{4}$. If $\beta_n \oplus \beta_{n-1} = 0$, then $c_n \oplus \beta_n = 0$ with probability $\frac{3}{4}$, thus $\bar{p}_{n+i,0} = \frac{3}{4} \times 1 + \frac{1}{4} \times (1 - 2^{-i+1}) =$ $1 - \frac{1}{4} \times 2^{-i+1}$. If $\beta_n \oplus \beta_{n-1} = 1$, then $c_n \oplus \beta_n = 0$ with probability $\frac{1}{4}$, thus $\bar{p}_{n+i,1} = \frac{1}{4} \times 1 + \frac{3}{4} \times (1 - 2^{-i+1}) = 1 - \frac{3}{4} \times 2^{-i+1}$. Then the difference $\Delta \bar{p}_{n+i} = \bar{p}_{n+i,0} - \bar{p}_{n+i,1} = 2^{-i}$ for i > 0.

The above two theorems provide the guidelines to recover the key of Phelix. However, these two theorems deal with the ideal cases in which there is only one bit difference between ϕ and ϕ' , and β is assumed to be random. In the attacks,

| j | p | j | p | j | p | j | p |
|---|--------|----|--------|----|--------|----|--------|
| 0 | 0.9997 | 8 | 1.0000 | 16 | 0.5001 | 24 | 0.9161 |
| 1 | 0.9998 | 9 | 0.0000 | 17 | 0.4348 | 25 | 0.9470 |
| 2 | 0.9999 | 10 | 0.5000 | 18 | 0.5000 | 26 | 0.9673 |
| 3 | 0.9999 | 11 | 0.4375 | 19 | 0.5486 | 27 | 0.9803 |
| 4 | 1.0000 | 12 | 0.5000 | 20 | 0.6366 | 28 | 0.9883 |
| 5 | 1.0000 | 13 | 0.4492 | 21 | 0.7283 | 29 | 0.9931 |
| 6 | 1.0000 | 14 | 0.5000 | 22 | 0.8083 | 30 | 0.9960 |
| 7 | 1.0000 | 15 | 0.4273 | 23 | 0.8708 | 31 | 0.9977 |

Table 2.1: The probability that $B_3^{(i+1),j} \oplus B_3^{\prime(i+1),j} = 0$ for $P_i \oplus P_i' = 1$

we deal with the complicated situation where each bit of $\phi \oplus \phi'$ is biased, and β is a fixed number. The value of each bit of β will affect the distribution of those more significant bits of $(\phi + \beta) \oplus (\phi' + \beta)$ in a complicated way. In order to simplify the analysis, we will use simulations to obtain these relations in the attacks.

2.4 A Basic Key Recovery Attack on Phelix

We will first investigate the differential propagation in Phelix. Then we show how to recover the key of Phelix by observing the differential distribution of the keystream.

2.4.1 The bias in the differential distribution of keystream

Assume an attacker can choose an arbitrary value for the nonce, then a nonce can be used more than once. We introduce one-bit difference into the plaintext at the *i*th step, i.e., $P_i \neq P'_i$, and $P_i \oplus P'_i = 2^n$ $(31 \ge n \ge 0)$. Then we analyze the difference between $B_3^{(i+1)}$ and $B_3^{\prime(i+1)}$ (as indicated in Fig. 1). If all the carry bits are 0 (replacing all the additions with XORs), then the differences only appear at the 9th, 11th, 13th, 15th and 17th least significant bits between $B_3^{(i+1)}$ and $B_3^{\prime(i+1)}$. Because of the carry bits, the differential distribution becomes complicated. We run the simulation and use the randomly generated $Y_k^{(i)}$ $(4 \ge k \ge 0)$, P_i , $X_{i,1}$ in the simulation. With 2^{30} plaintext pairs, we obtain the distribution of $B_3^{(i+1)} \oplus B_3^{\prime(i+1)}$ in Table 1.

From Table 2.1, we see that the distribution of $B_3^{(i+1)} \oplus B_3^{\prime(i+1)}$ is heavily

biased. For example, $B_3^{(i+1),8} = B_3^{\prime(i+1),8}$ with probability close to 1, while $B_3^{(i+1),9} = B_3^{\prime(i+1),9}$ with probability close to 0. Note that $T_0^{(i+1)} = A_0^{(i+1)} \oplus (B_3^{(i+1)} + X_{i+1,0})$, according to Theorem 2.2, the distribution of $T_0^{(i+1)} \oplus T_0^{\prime(i+1)}$ will be affected by the value of $X_{i+1,0}^8 \oplus X_{i+1,0}^9$. By observing the distribution of $S_{i+1} \oplus B_{i+1}^{\prime}$ will be affected by the value of $X_{i,0}^8 \oplus X_{i,0}^9$. By observing the distribution of $S_{i+1} \oplus S_{i+1}^{\prime}$, it may be possible to determine the value of $X_{i,0}^8 \oplus X_{i,0}^9$. Shifting the one-bit difference between P_i and P_i^{\prime} , we may determine other values of $X_{i,0}^{j+1} \oplus X_{i,0}^j$ for $30 \ge j \ge 0$, and thus recover the key $X_{i,0}$. After recovering eight consequtive $X_{i,0}$, the 256-bit key is immediately known.

The above analysis gives a brief idea of the attack. However, the actual attacks are quite complicated due to the interference of many differences. It is very tedious to derive exactly how the distribution of $S_{i+1} \oplus S'_{i+1}$ is affected by the value of $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$. On the other hand, it is easy to search for the relation with simulations. In the following, we carried out the simulation to find out the relation between the value of $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$ and the distribution of $S_{i+1} \oplus S'_{i+1}$.

Let two plaintexts differ only in the *i*th word, and $P_i \oplus P'_i = 1$. We use the randomly generated $Y_k^{(i)}$ $(4 \ge k \ge 0)$, P_i , $X_{i,1}$, $X_{i+1,0}$, $Z_4^{(i-3)}$ in the simulation. Denote $p_{j,0}^n$ as the probability that $S_{i+1}^n \oplus S_{i+1}^{\prime n} = 0$ when $X_{i,0}^{j+1} \oplus X_{i,0}^j = 0$. And denote $p_{j,1}^n$ as the probability that $S_{i+1}^n \oplus S_{i+1}^{\prime n} = 0$ when $X_{i,0}^{j+1} \oplus X_{i,0}^j = 1$. Let $\Delta \tilde{p}_j^n = (p_{j,0}^n - p_{j,1}^n) \times \frac{N}{\sigma}$, where N denotes the number of plaintext pairs, and $\sigma = \frac{\sqrt{N}}{2}$. Assume that the values of $p_{j,0}^n$ and $p_{j,1}^n$ are close to $\frac{1}{2}$. If $\Delta \tilde{p}_j^n > 4$, it means that the difference between $p_{j,0}^n$ and $p_{j,1}^n$ is larger than 4σ , then the value of $X_{i,0}^{j+1} \oplus X_{i,0}^j$ can be determined correctly with high probability. For every value of the two bits $X_{i,0}^{j+1}$ and $X_{i,0}^j$, we use 2^{28} pairs to generate $S_{i+1} \oplus S_{i+1}^{\prime}$, then compute $p_{j,0}^n$ and $p_{j,1}^n$. Thus $N = 2^{29}$, and $\sigma = 2^{13.5}$. We list the large values of $\Delta \tilde{p}_j^n$ below:

$$\begin{split} & \text{For } j = 9, \, \Delta \tilde{p}_{10}^{13} = 55.7 \; . \\ & \text{For } j = 10, \, \Delta \tilde{p}_{10}^{13} = 133.9 \; . \\ & \text{For } j = 14, \, \Delta \tilde{p}_{14}^{17} = 51.5 \; . \\ & \text{For } j = 15, \, \Delta \tilde{p}_{15}^{19} = -9.1, \, \Delta \tilde{p}_{15}^{22} = 14.9, \, \Delta \tilde{p}_{15}^{23} = -15.7 \; . \\ & \text{For } j = 16, \, \Delta \tilde{p}_{16}^{19} = -50.8, \, \Delta \tilde{p}_{16}^{21} = 62.0, \, \Delta \tilde{p}_{16}^{22} = 97.7, \, \Delta \tilde{p}_{16}^{23} = -106.6, \\ & \Delta \tilde{p}_{16}^{25} = 11.8, \, \Delta \tilde{p}_{16}^{26} = 16.0, \, \Delta \tilde{p}_{16}^{27} = -17.4 \; . \\ & \text{For } j = 17, \, \Delta \tilde{p}_{17}^{21} = 77.4, \, \Delta \tilde{p}_{17}^{22} = 145.3, \, \Delta \tilde{p}_{17}^{23} = -171.6, \, \Delta \tilde{p}_{17}^{25} = 12.3, \\ & \Delta \tilde{p}_{17}^{26} = 28.5, \, \Delta \tilde{p}_{17}^{27} = -30.4 \; . \\ & \text{For } j = 18, \, \Delta \tilde{p}_{18}^{21} = 80.2, \, \Delta \tilde{p}_{18}^{22} = 179.7, \, \Delta \tilde{p}_{18}^{23} = -241.7, \, \Delta \tilde{p}_{18}^{26} = 32.8, \\ & \Delta \tilde{p}_{18}^{27} = -43.7 \; . \\ & \text{For } j = 19, \, \Delta \tilde{p}_{19}^{22} = 139.6, \, \Delta \tilde{p}_{19}^{23} = -220.6, \, \Delta \tilde{p}_{19}^{26} = 19.0, \, \Delta \tilde{p}_{19}^{27} = -46.5 \end{split}$$

For
$$j = 20$$
, $\Delta \tilde{p}_{20}^{23} = -156.7$, $\Delta \tilde{p}_{20}^{25} = -5.7$, $\Delta \tilde{p}_{20}^{26} = 18.3$, $\Delta \tilde{p}_{20}^{27} = -30.6$.
For $j = 21$, $\Delta \tilde{p}_{21}^{25} = -6.8$, $\Delta \tilde{p}_{21}^{26} = 9.5$, $\Delta \tilde{p}_{20}^{27} = -28.5$.

The data given above show that the distribution of $S_{i+1} \oplus S'_{i+1}$ is strongly affected by the value of $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$.

2.4.2 Recovering the key

Note that in the above analysis, when we deal with a particular $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$, the other bits of $X_{i+1,0}$ are random. In the key recovery attack, the value of $X_{i+1,0}$ is fixed, so we need to consider the interference between the bits $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$.

We notice that there are many large biases related to $S_{i+1}^{23} \oplus S_{i+1}^{\prime 23}$. However, the values of $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$ (15 $\leq j \leq 20$) all have a significant effect on the distribution of $S_{i+1}^{23} \oplus S_{i+1}^{\prime 23}$. It is thus a bit complicated to determine the values of $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$ (15 $\leq j \leq 20$).

of $X_{i+1,0} \oplus X_{i+1,0}$ (15 $\leq j \leq 20$). In the following, we consider the bit $S_{i+1}^{17} \oplus S_{i+1}^{\prime 17}$. Its distribution is dominated by the value of $X_{i+1,0}^{15} \oplus X_{i+1,0}^{14}$. For every value of the two bits $X_{i+1,0}^{15}$ and $X_{i+1,0}^{14}$, we use 2^{30} pairs to generate $S_{i+1} \oplus S_{i+1}^{\prime}$, then compute $p_{14,0}^{17}$ and $p_{14,1}^{17}$. From the simulation, we found that $p_{14,0}^{17} = 0.50227$ and $p_{14,1}^{17} = 0.50117$. We denote the average of $p_{14,0}^{17}$ and $p_{14,1}^{17}$ as \bar{p}_{14}^{17} , i.e., $\bar{p}_{14}^{17} = \frac{p_{14,0}^{17} + p_{14,1}^{17}}{2} = 0.50172$. Running a similar simulation, we found that $p_{13,0}^{17} = 0.50175$ and $p_{13,1}^{17} = 0.50169$. For all the $j \neq 13, j \neq 14$, we found that $p_{j,0}^{17} \approx \bar{p}_{14}^{17}$ and $p_{j,1}^{17} \approx \bar{p}_{14}^{17}$. The value of $X_{i+1,0}^{15} \oplus X_{i+1,0}^{14}$ is recovered as follows: from the keystreams, we compute the fraction for which $S_{i+1}^{17} \oplus S_{i+1}^{\prime 17} = 0$. If it is larger than \bar{p}_{14}^{17} , then the value of $X_{i+1,0}^{15} \oplus X_{i+1,0}^{14}$ is considered to be 0; otherwise the value of $X_{i+1,0}^{15} \oplus X_{i+1,0}^{14}$ is considered to be 1.

We now compute the number of plaintext pairs required to determine the value of $X_{i+1,0}^{15} \oplus X_{i+1,0}^{14}$. Suppose that N pairs of plaintexts are used. The standard deviation is $\sigma = \sqrt{N \times \bar{p}_{14}^{17} \times (1 - \bar{p}_{14}^{17})}$. To determine the value of $X_{i+1,0}^{15} \oplus X_{i+1,0}^{14}$ with success rate 0.99, we require that $N \times ((p_{14,0}^{17} - p_{14,1}^{17}) - (p_{13,0}^{17} - p_{13,1}^{17})) > 4.66 \times \sigma$ (The cumulative distribution function of the normal distribution gives value 0.99 at the point 2.33 σ). Thus we require that $N > 2^{22.27}$.

We used the Phelix C source code submitted to eSTREAM in the experiments. However, there is a bug in the C source code. The output is given as $S_i = Y_4^{(i)} + Z_4^{(i-3)}$ instead of $S_i = Y_4^{(i)} + Z_4^{(i-4)}$ which is specified in the chapter. The Phelix C code with the bug being fixed was used in the experiments.

Experiment 2.1. The experiment is to recover the value of $X_{1,0}^{15} \oplus X_{1,0}^{14}$. Each

plaintext has two words P_0 and P_1 . For each plaintext pair, the two words differ only in the least significant bit of P_0 . N plaintext pairs are used for each key to determine the value of $X_{1,0}^{15} \oplus X_{1,0}^{14}$ as follows: if the fraction of cases for which $S_1^{17} \oplus S_1'^{17} = 0$ is larger than $\bar{p}_{14}^{17} = 0.50172$, then the value of $X_{1,0}^{15} \oplus X_{1,0}^{14}$ is considered to be 0; otherwise the value of $X_{1,0}^{15} \oplus X_{1,0}^{14}$ is considered to be 1. A random nonce was used for each plaintext pair. We tested 200 keys in the experiment. For $N = 2^{22.3}$, the values of $X_{1,0}^{15} \oplus X_{1,0}^{14}$ of 183 keys are determined correctly. For $N = 2^{25}$, the values of $X_{1,0}^{15} \oplus X_{1,0}^{14}$ of 192 keys are determined correctly.

Experiment 2.1 shows that the value of $X_{1,0}^{15} \oplus X_{1,0}^{14}$ can be determined successfully by introducing a difference in the least significant bit of P_0 , but with a higher error rate. The reason is that other bits of $X_{1,0}$ affects the determination of $X_{1,0}^{15} \oplus X_{1,0}^{14}$ in a subtle way.

We now proceed to recover the other bits of $X_{1,0}$. By rotating the onebit difference between P_0 and P'_0 , and using the same threshold value, we can determine the value of $X_{1,0}^{j+1} \oplus X_{1,0}^{j}$ for $2 \le j \le 3$, $5 \le j \le 10$ and $14 \le j \le 28$.

Thus we are able to recover 23 bits of information on each $X_{i,0}$. For the 256-bit key of Phelix, we are able to recover $23 \times 8 = 184$ bits of the key with success rate about $\frac{192}{200} = 0.96$. The amount of plaintext pairs required in the attack is about $2^{25} \times 32 \times 8 = 2^{33}$.

We need to improve the above attack in two approaches: recovering more key bits and improving the success rate. The direct approach is to adjust the threshold value for each key bit position. In the following, we illustrate a more advanced approach which recovers the values of $Z_4^{(i)}$ before recovering the key.

2.5 Improving the Attack on Phelix

In the above attack, we use a random nonce for each plaintext pair, i.e., every nonce is used twice with the same key. When the nonce is used many times with the same key, we can introduce the difference at P_i and recover the value of Z_4^{i-3} by observing the distribution of $S_{i+1} \oplus S'_{i+1}$. Then we proceed to recover $X_{i+1,0}$.

2.5.1 Recovering $Z_4^{(i)}$

We introduce the difference to the least significant bit of $P_i (P_i \oplus P'_i = 1)$. A simulation is carried out to determine the distribution of $Y_4^{(i+1)} \oplus Y_4^{\prime(i+1)}$. We use the randomly generated $Y_k^{(i)}$ $(4 \ge k \ge 0)$, P_i , $X_{i,1}$, $X_{i+1,0}$ in the simulation. Denote \dot{p}_n as the probability that $Y_4^{(i+1),n} \oplus Y_4^{\prime(i+1),n} = 0$. With 2^{30} pairs, we obtain the values of \dot{p}_n in Table 2.2.

| j | $\dot{p}_{j} - 0.5$ | j | $\dot{p}_{j} - 0.5$ | j | $\dot{p}_{j} - 0.5$ | j | $\dot{p}_{j} - 0.5$ |
|---|---------------------|----|---------------------|----|---------------------|----|---------------------|
| 0 | 0.03326 | 8 | 0.00003 | 16 | -0.00003 | 24 | 0.00046 |
| 1 | 0.12983 | 9 | 0.03517 | 17 | 0.00268 | 25 | 0.05926 |
| 2 | 0.20291 | 10 | 0.00002 | 18 | -0.00001 | 26 | 0.15064 |
| 3 | -0.27754 | 11 | 0.00001 | 19 | -0.00266 | 27 | -0.24028 |
| 4 | -0.00005 | 12 | 0.00000 | 20 | -0.00004 | 28 | 0.00001 |
| 5 | 0.05663 | 13 | 0.02293 | 21 | 0.02276 | 29 | 0.05770 |
| 6 | -0.15327 | 14 | -0.00001 | 22 | 0.07434 | 30 | 0.15508 |
| 7 | -0.00001 | 15 | -0.00001 | 23 | -0.14414 | 31 | -0.24907 |

Table 2.2: The probability that $Y_4^{(i+1),j} \oplus Y_4^{\prime(i+1),j} = 0$ for $P_i \oplus P_i' = 1$

From Table 2.2, we notice that $Y_4^{(i+1)} \oplus Y_4'^{(i+1)}$ is heavily biased. For example, $Y_4^{(i+1),2} = Y_4'^{(i+1),2}$ with probability about 0.70291, while $Y_4^{(i+1),3} = Y_4'^{(i+1),3}$ with probability about 0.22246. Note that $S_{i+1} = Y_4^{(i+1)} \oplus Z_4^{(i-3)}$, according to Theorem 2.2, the distribution of $S_{i+1} \oplus S_{i+1}'$ is affected by the value of $Z_4^{(i-3),3} \oplus Z_4^{(i-3),2}$. Next we carry out simulations to characterize this relation.

We use the randomly generated $Y_k^{(i)}$ $(4 \ge k \ge 0)$, P_i , $X_{i,1}$, $X_{i+1,0}$, $Z_4^{(i-3)}$ in the simulation. The one-bit difference is introduced to P_i , i.e., $P_i \oplus P'_i = 2^j$. Denote $\ddot{p}_{j,0}^n$ as the probability that $S_{i+1}^n \oplus S_{i+1}^{\prime n} = 0$ when $Z_4^{(i-3),j+1} \oplus Z_4^{(i-3),j} = 0$. And denote $\ddot{p}_{j,1}^n$ as the probability that $S_{i+1}^n \oplus S_{i+1}^{\prime n} = 0$ when $Z_4^{(i-3),j+1} \oplus Z_4^{(i-3),j+1} \oplus Z_4^{(i-3),j} = 1$. For each value of $Z_4^{(i-3),3}$ and $Z_4^{(i-3),2}$, we use 2²⁸ plaintext pairs. We find that $\ddot{p}_{2,0}^5 = 0.5461$ and $\ddot{p}_{2,1}^5 = 0.5193$. The large difference between $\ddot{p}_{2,0}^5$ and $\ddot{p}_{2,1}^5$ shows that the value of $Z_4^{(i-3),3} \oplus Z_4^{(i-3),2}$ can be determined with success rate 0.999 with about 2^{13.9} plaintext pairs (The cumulative distribution function of the normal distribution gives value 0.999 at the point 3.1 σ).

The above approach is able to recover $Z_4^{(0)}$, but the success rate is not that high according to our experiment. In the following, we use a new approach to determine $Z_4^{(0)}$. To reduce the interference between the bits of $Z_4^{(i-3)}$, we recover the least significant bit of $Z_4^{(i-3)}$ first, then proceed to recover the more significant bits bit-by-bit.

We start with determining the value of $Z_4^{(i-3),0}$. Let $P_i \oplus P'_i = 1$. Running the simulation with 2^{28} plaintext pairs, we found that $\ddot{p}_{-1,0}^2 = 0.70296$, and $\ddot{p}_{-1,1}^2 = 0.65422$ (let $Z_4^{(i-3),-1} = 0$). To determine value of $Z_4^{(i-3),0}$ with success rate 0.999, we need about $2^{12.0}$ plaintext pairs.

Experiment 2.2. This experiment is to determine the value of $Z_4^{(0),0}$. Each plaintext has five random words P_i $(0 \le i \le 4)$. For each plaintext pair, the difference is only in the least significant bit of P_3 . N plaintext pairs are used for each key/nonce pair to determine the value of $Z^{(0),0}$ as follows: if the rate that $S_4^2 \oplus S_4'^2 = 0$ is larger than $\frac{\ddot{p}_{-1,0}^2 + \ddot{p}_{-1,1}^2}{2} = \frac{0.70296 \pm 0.65422}{2} = 0.6786$, then the value of $Z_4^{(0),0}$ is considered to be 0; otherwise the value of $Z_4^{(0),0}$ is considered to be 1. We tested 1000 key/nonce pairs in the experiment. For $N = 2^{12}$, the values of $Z_4^{(0),0}$ of 998 key/nonce pairs are determined correctly. For $N = 2^{13}$, the values of $Z_4^{(0),0}$ of all the key/nonce pairs are determined correctly.

After recovering the value of $Z_4^{(i-3),0}$, we proceed to recover the values of the other bits of $Z_4^{(i-3)}$. Let $Z_4^{(i-3),(n-1\cdots0)}$ denote the *n* least significant bits of $Z^{(i-3)}$, i.e., $Z_4^{(i-3),(n-1\cdots0)} = Z^{(i-3)} \mod 2^n$. Let the difference be introduced to the *k*th least significant bit of P_i , i.e., $P_i \oplus P'_i = 2^k$. Denote $\dot{p}_{Z_4^{(i-3),(n-1\cdots0)}}^{k,j,0}$ as the probability that the value of the *j*th bit of $(S_{i+1} - Z_4^{(i-3),(n-1\cdots0)}) \oplus (S'_{i+1} - Z_4^{(i-3),(n-1\cdots0)})$ is 0 when $Z_4^{(i-3),n} = 0$. Denote $\dot{p}_{Z_4^{(i-3),(n-1\cdots0)}}^{k,j,1}$ as the probability that the value of $(S_{i+1} - Z_4^{(i-3),(n-1\cdots0)}) \oplus (S'_{i+1} - Z_4^{(i-3),(n-1\cdots0)})$ is 0 when $Z_4^{(i-3),n} = 1$. If the value of $Z_4^{(i-3),(n-1\cdots0)}$ is determined correctly, then $\dot{p}_{Z_4^{(i-3),(n-1\cdots0)}} = \dot{p}_0^{k,j,0}$, and $\dot{p}_{Z_4^{(i-3),(n-1\cdots0)}} = \dot{p}_0^{k,j,1}$. This property is important for recovering $Z_4^{(i-3)}$.

Let $P_i \oplus P'_i = 2$. We use 2^{28} plaintext pairs in the simulation. We found that $\dot{p}_0^{1,3,0} = 0.66469$ and $\dot{p}_0^{1,3,1} = 0.60220$. It shows that when $Z_4^{(i-3),0} = 0$, if the rate that $S_{i+1}^3 \oplus S_{i+1}'^3 = 0$ is larger than $\frac{0.66469+0.60220}{2} = 0.63345$, then the value of $Z_4^{(i-3),1}$ is determined to be 0; otherwise the value of $Z_4^{(i-3),1}$ is determined to be 1. We need about $2^{11.3}$ plaintext pairs to determine the value of $Z_4^{(i-3),1}$ correctly with success rate 0.999. Using the Phelix code in the experiment, we tested 1000 random key/nonce pairs satisfying $Z_4^{(0),0} = 0$, and 2^{12} plaintext pairs are used for each key/nonce pair with the difference $P_3 \oplus P'_3 = 2$. We found that all the 1000 values of $Z_4^{(0),1}$ are determined correctly. If $Z_4^{(i-3),0} = 1$, we observe the third least significant bit of $(S_{i+1} - 1) \oplus (S'_{i+1} - 1)$, and we can determine the value of $Z_4^{(0),1} = 0$ with success rate 0.999 with about $2^{11.3}$ plaintext pairs.

Let $P_i \oplus P'_i = 2^2$, we are able to determine the value of $Z_4^{(i-3),2}$ by observing the fourth least significant bit of $(S_{i+1} - Z_4^{(i-3),(1\cdots 0)}) \oplus (S'_{i+1} - Z_4^{(i-3),(1\cdots 0)})$. In general, let $P_i \oplus P'_i = 2^j$, then we are able to determine the value of $Z_4^{(i-3),j}$ by observing the (j+2)th least significant bit of $(S_{i+1} - Z_4^{(i-3),(j-1\cdots 0)}) \oplus (S'_{i+1} -$ $Z_4^{(i-3),(j-1\cdots 0)}$) with about 2^{12} plaintext pairs. Thus we are able to recover $Z_4^{(i-3)}$ (except the values of $Z_4^{(i-3),30}$ and $Z_4^{(i-3),31}$) with success rate very close to 1. The number of plaintext pairs required in the above attack is about $2^{12} \times 30 \approx 2^{17}$.

2.5.2 Recovering $X_{i+1,0}$

After recovering $Z_4^{(i-3)}$ (except $Z_4^{(i-3),31}$ and $Z_4^{(i-3),30}$), we know the value of $(S_{i+1} - Z_4^{(i-3),(29\cdots 0)}) \oplus (S'_{i+1} - Z_4^{(i-3),(29\cdots 0)})$. Thus we know the value of $Y^{(i+1),j} \oplus Y'^{(i+1),j}$ ($0 \le j \le 30$). Then we are able to recover $X_{i+1,0}$ more efficiently.

Let two plaintexts differ only in the *i*th word. And let $P_i \oplus P'_i = 1$. We use the randomly generated $Y_k^{(i)}$ $(4 \ge k \ge 0)$, P_i , $X_{i,1}$, $X_{i+1,0}$, $Z_4^{(i-3)}$ in the simulation. For every value of the two bits $X_{i,0}^{j+1}$ and $X_{i,0}^{j}$, we use 2^{28} plaintext pairs to generate $Y_4^{(i+1)} \oplus Y_4^{\prime(i+1)}$, then compute $p_{j,0}^n$ and $p_{j,1}^n$ (suppose that $S_{i+1} = Y_4^{(i+1)}$ since $Z_4^{(i-3)}$ is known). Thus $N = 2^{29}$, and $\sigma = 2^{13.5}$. We list the following two large biases $\Delta \tilde{p}_j^n$:

For j = 9, $\Delta \tilde{p}_{9}^{13} = 144.1$ For j = 10, $\Delta \tilde{p}_{10}^{13} = 362.12$

We use $\Delta \tilde{p}_{9}^{13}$ and $\Delta \tilde{p}_{10}^{13}$ in the attack. Note that the values of $X_{i+1,0}^{10} \oplus X_{i+1,0}^{9}$ and $X_{i+1,0}^{11} \oplus X_{i+1,0}^{10}$ both affect the distribution of $Y_4^{(i+1),13} \oplus Y_4^{\prime(i+1),13}$. We carried out a simulation to determine how the value of $X_{i+1,0}^{9}$ affects the value of $Y_4^{(i+1),13}$. With 2³⁰ chosen plaintext pairs, if $X_{i+1,0}^{9} = 0$, then $p_{0,00}^{13} = 0.53033$, $p_{0,11}^{13} = 0.52334$, $p_{0,01}^{13} = 0.51946$, $p_{0,10}^{13} = 0.51864$; if $X_{i+1,0}^{9} = 1$, then $p_{0,00}^{13} = 0.52334$, $p_{0,11}^{13} = 0.53030$, $p_{0,01}^{13} = 0.51861$, $p_{0,10}^{13} = 0.51948$. We thus let $p_{0,0}^{13} = 0.52334$, and $p_{0,1}^{13} = \frac{0.51946+0.51948}{2} = 0.51947$. About 2^{19.3} plaintext pairs are required to determine the value of $X_{i+1,0}^{11} \oplus X_{i+1,0}^{10}$ with success rate 0.999.

Experiment 2.3. Suppose that the value of $Z_4^{(0)}$ is known. This experiment is to determine the value of $X_{4,0}^{11} \oplus X_{4,0}^{10}$. Each plaintext has five random words P_i $(0 \le i \le 4)$. For each plaintext pair, those five words differ only in the least significant bit of P_3 . N plaintext pairs are used for each key/nonce pair to determine the value of $X_{4,0}^{11} \oplus X_{4,0}^{10}$ as follows: if the rate that $Y_4^{13} \oplus Y_4^{/13} = 0$ is larger than $\frac{0.52334+0.51947}{2} = 0.52140$, then the value of $X_{4,0}^{11} \oplus X_{4,0}^{10}$ is considered to be 0; otherwise the value of $X_{4,0}^{11} \oplus X_{4,0}^{10}$ is considered to be 1. We tested 1000 key/nonce pairs in the experiment. For $N = 2^{19.3}$, 948 values of 1000 $X_{4,0}^{11} \oplus X_{4,0}^{10}$ are determined correctly. We change the threshold value 0.52140 to 0.52035, then 970 values of 1000 $X_{4,0}^{11} \oplus X_{4,0}^{10}$ are determined correctly for $N = 2^{20}$, 976 values

are determined correctly for $N = 2^{21}$, 990 values are determined correctly for $N = 2^{22}$.

Experiment 2.3 shows that the value of $X_{4,0}^{11} \oplus X_{4,0}^{10}$ can be determined successfully by introducing a difference to the least significant bit of P_3 . With 2^{22} chosen pairs, we are able to determine the value of $X_{4,0}^{11} \oplus X_{4,0}^{10}$ with success rate about 0.99.

Then we shift the one-bit difference to recover the values of $X_{1,0}^{j+1} \oplus X_{1,0}^{j}$ for $2 \leq j \leq 28$. The threshold value needs to be modified for different values of j. The number of plaintext pairs and the threshold value required to recover the value of each $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$ ($2 \leq j \leq 28$) are given in Table 2.3. Each value n in the second column indicates that the difference is introduced in the nth least significant bit of P_i . Each value n in the third column shows that the nth least significant bit of $Y_4^{(i+1)}$ is used in the attack. Note that according to Experiment 2.3, the threshold values should be slightly adjusted to achieve a high success rate.

The reason that the values of $X_{4,0}^{j+1} \oplus X_{4,0}^{j}$ cannot be recovered for $j \geq 29$ is that the value of $X_{1,0}^{j+1} \oplus X_{1,0}^{j}$ cannot affect the distribution of $S_1^{j+3} \oplus S_1'^{j+3}$ since $S_1 \oplus S_1'$ is a 32-bit word. The reason that the number of plaintexts required for j = 9 is relatively small is that the difference for j = 13 is introduced to the most significant bit of the word P_3 , thus it causes less difference propagation, and results in a larger bias in the keystream.

Note that the most significant bit of $Y_4^{(i+1)} \oplus Y_4^{\prime(i+1)}$ is not known since $Z_4^{i-1,31}$ and $Z_4^{i-1,30}$ are not recovered. Thus to determine the value of $X_{1,0}^{29} \oplus X_{1,0}^{28}$, we need to consider the most significant bit of $(S_{i+1} - Z_4^{(i-3),(29\cdots 0)}) \oplus (S'_{i+1} - Z_4^{(i-3),(29\cdots 0)})$. The threshold value needs to be changed to 0.51128; and the number of plaintext pairs required is $2^{22.1}$.

After recovering the values of $X_{1,0}^{j+1} \oplus X_{1,0}^{j}$ for $2 \leq j \leq 28$, we proceed to determine the value of $X_{i+1,0}^{0}$, $X_{i+1,0}^{1}$ and $X_{i+1,0}^{2}$.

We start with recovering $X_{i+1,0}^0$. Let $P_i \oplus P'_i = 2^{21}$. Running the simulation with 2^{28} plaintext pairs, we found that $p_{21,0}^2 = 0.51596$, $p_{21,1}^2 = 0.50355$. Thus $2^{15.93}$ plaintext pairs are needed to determine the value of $X_{i+1,0}^0$ with success rate 0.999. Using the Phelix code in the experiment, we introduce the difference $P_3 \oplus P'_3 = 2^{21}$, and set the threshold value as $\frac{0.51596+0.50355}{2} = 0.50975$. We tested 1000 key/nonce pairs in the experiment. With 2^{16} plaintext pairs, all the values of the 1000 $X_{4,0}^0$ are determined correctly.

After determining the value of $X_{i+1,0}^0$, we determine the value of $X_{i+1,0}^1$ as follows. The simulation shows that the value of $X_{i+1,0}^1$ can be determined only when $X_{i+1,0}^0 = 0$. For $X_{i+1,0}^0 = 0$, we set the difference as $P_i \oplus P'_i = 2^{22}$. With

| j | Difference | Bit | Threshold | Plaintext |
|----|------------|------------------|-----------|------------|
| | position | position | value | Pairs |
| | in P_i | in $Y_4^{(i+1)}$ | | |
| 2 | 24 | 5 | 0.51101 | $2^{24.4}$ |
| 3 | 25 | 6 | 0.51110 | $2^{23.1}$ |
| 4 | 26 | 7 | 0.51120 | $2^{22.5}$ |
| 5 | 27 | 8 | 0.51125 | $2^{22.3}$ |
| 6 | 28 | 9 | 0.51091 | $2^{22.4}$ |
| 7 | 29 | 10 | 0.51116 | $2^{23.6}$ |
| 8 | 30 | 11 | 0.51562 | $2^{20.7}$ |
| 9 | 31 | 12 | 0.54353 | $2^{18.4}$ |
| 10 | 0 | 13 | 0.52141 | $2^{19.3}$ |
| 11 | 1 | 14 | 0.52099 | $2^{19.3}$ |
| 12 | 2 | 15 | 0.51850 | $2^{19.5}$ |
| 13 | 3 | 16 | 0.50998 | $2^{21.3}$ |
| 14 | 4 | 17 | 0.51107 | $2^{21.9}$ |
| 15 | 5 | 18 | 0.51128 | $2^{22.2}$ |
| 16 | 6 | 19 | 0.51129 | $2^{22.2}$ |
| 17 | 7 | 20 | 0.51131 | $2^{22.2}$ |
| 18 | 8 | 21 | 0.51128 | $2^{22.1}$ |
| 19 | 9 | 22 | 0.51117 | $2^{21.7}$ |
| 20 | 10 | 23 | 0.51149 | $2^{22.2}$ |
| 21 | 11 | 24 | 0.51172 | $2^{22.0}$ |
| 22 | 12 | 25 | 0.51187 | $2^{22.0}$ |
| 23 | 13 | 26 | 0.51191 | $2^{22.0}$ |
| 24 | 14 | 27 | 0.51185 | $2^{22.1}$ |
| 25 | 15 | 28 | 0.51129 | $2^{22.2}$ |
| 26 | 16 | 29 | 0.51129 | $2^{22.1}$ |
| 27 | 17 | 30 | 0.51131 | $2^{22.2}$ |
| 28 | 18 | 31 | 0.51130 | $2^{22.1}$ |

Table 2.3: The number of plaintext pairs for recovering $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$

 2^{28} chosen plaintext pairs, we found that if $X_{i+1,0}^1 = 0$, then $Y_4^{(i+1),3} = 0$ with rate 0.51528; otherwise $Y_4^{(i+1),3} = 0$ with rate 0.50459. With $2^{16.4}$ plaintext pairs, the value of $X_{i+1,0}^1$ can be determined with success rate 0.999. Using the Phelix code in the experiment, we introduce the difference $P_3 \oplus P'_3 = 2^{22}$, and set the threshold value as $\frac{0.51528+0.50459}{2} = 0.50994$. We tested 1000 key/nonce pairs with $X_{4,0}^0 = 0$ in the experiment. With $2^{16.4}$ plaintext pairs, all the values of the 1000 $X_{4,0}^1$ are determined correctly. It shows that the value of $X_{i+1,0}^1$ can be determined successfully if $X_{4,0}^0 = 0$.

We continue to recover the value of $X_{i+1,0}^2$. We introduce difference to the 15th least significant bit of P_i , and observe the distribution of $Y_4^{(i-3),4}$. We carry out a simulation with 2^{31} plaintext pairs with $P_3 \oplus P'_3 = 2^{15}$. 2^{31} plaintext pairs are used for each value of $X_{i+1,0}^1 X_{i+1,0}^0$. When $X_{i+1,0}^0 = 0$, if $X_{i+1,0}^1 = 0$, the rates that $Y_4^{(i-3),4} = 0$ for $X_{i+1,0}^2 = 0$ and $X_{i+1,0}^2 = 1$ are 0.53106 and 0.52613, respectively; if $X_{i+1,0}^1 = 1$, the rates that $Y_4^{(i-3),4} = 0$ for $X_{i+1,0}^2 = 0$ and $X_{i+1,0}^2 = 1$ are 0.52318 and 0.52315, respectively. It shows that the value $X_{i+1,0}^2$ can only be determined if the values of $X_{i+1,0}^1$ and $X_{i+1,0}^1$ are both zero, and $2^{18.6}$ plaintext pairs are required to achieve the success rate 0.999.

In the above attacks, we recovered 28.75 bits of $X_{i+1,0}$: $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$ for $2 \leq j \leq 28$, $X_{i+1,0}^{0}$, $X_{i+1,0}^{1}$ (only if $X_{i+1,0}^{0} = 0$), and $X_{i+1,0}^{1}$ (only if $X_{i+1,0}^{0} = 0$ and $X_{i+1,0}^{1} = 0$). 27 bits of $X_{i+1,0}^{j+1} \oplus X_{i+1,0}^{j}$ ($2 \leq j \leq 28$) can be determined according to Table 3. From Experiment 2.3, we know that if we adjust slightly the threshold value, and use about $2^{2.7}$ times the number of plaintext pairs than that given in Table 3, the success rate is about 0.99. The number of plaintext pairs required to determine these 27 bits is thus about $27 \times 2^{22.2} \times 2^{2.7} = 2^{29.7}$. The number of plaintext pairs to determine $X_{i+1,0}^{0}$, $X_{i+1,0}^{1}$ and $X_{i+1,0}^{2}$ is small compared to $2^{29.7}$. The attack to recover 28.75 bits of $X_{i+1,0}$ requires thus about $2^{32.7}$ chosen plaintext pairs.

After recovering eight consecutive $X_{i+1,0}$, we recovered $28.75 \times 8 = 230$ key bits. To recover the 256-bit key, the number of operations required is about $2^{256-230+\binom{27\times8}{(0.01\times27\times8)}} = 2^{41.5}$.

The number of chosen plaintext pairs required in the attack is about $2^{29.7} \times 8 = 2^{32.7}$. The length of each plaintext ranges from 5 to 13 words. Thus the total amount of chosen plaintext required is about $2 \times 2^{32.7} \times \frac{5+13}{2} \approx 2^{37}$ words. (The number of plaintext pairs needed to recover 8 consecutive $Z_4^{(i)}$ value is about $2^{17} \times 8 = 2^{25}$. It is small compared to $2^{32.7}$).

2.6 How to strengthen Helix and Phelix

In Helix and Phelix, the plaintext is used to affect the internal state of the cipher. In order to achieve a high encryption speed, each plaintext word affects the keystream without passing through sufficient confusion and diffusion layers. This is the intrinsic weakness in the structure of Helix and Phelix. In the following, we provide a method to reduce the effect of such weakness.

The security of the encryption of Helix and Phelix can be improved significantly if a secure one-way function is used to generate the initial state of the cipher from the key and nonce. Then even if the internal state of one particular nonce is recovered, the impact on the security of the encryption is very limited since the key of the cipher is not affected. We believe that such an approach can be applied to improve the security of all the ciphers that use the plaintext to affect the internal state.

However, we must point out that such an approach does not improve significantly the security of the MAC in Helix and Phelix. Once an internal state is recovered, the attacker can forge many messages related to that particular nonce.

2.7 Conclusion

Phelix is vulnerable to a key recovery attack when chosen nonces and chosen plaintexts are used. The computational complexity of the attack is much less than that of the attack against Helix. Our attack shows that Phelix fails to strengthen Helix in this respect.

We believe that one necessary requirement for a secure general-purpose stream cipher is that the key of the cipher should not be recoverable even if the attacker can control the generation of the nonce. We thus consider Phelix as insecure. Note that Muller has pointed out the impact of the key recovery attack on the security of Helix in detail [96]. The same comments apply to Phelix.

Chapter 3

Exploiting Characteristics of Addition II Fast Correlation Attack on the Stream Cipher ABC v2

Abstract. ABC v2 is a software-efficient stream cipher with a 128-bit key. In this chapter, we apply a fast correlation attack to break ABC v2 with weak keys. There are about 2^{96} weak keys in ABC v2. The complexity to identify a weak key and to recover the internal state of a weak key is low: identifying one weak key from about 2^{32} random keys requires 6460 keystream bytes and $2^{13.5}$ operations for each random key. Recovering the internal state of a weak key requires about $2^{19.5}$ keystream bytes and $2^{32.8}$ operations. A similar attack can be applied to break ABC v1 with much lower complexity than the previous attack on ABC v1.

3.1 Introduction

ABC [4] is a stream cipher submitted to the ECRYPT eStream project. It is one of the fastest submissions with encryption speed about 3.5 cycles/byte on the Intel Pentium 4 microprocessor.

ABC v1 was broken by Berbain and Gilbert [14] (later by Khazaei [81]). Their divide-and-conquer attack on ABC exploits the short length (63 bits) of the LFSR in the component A and the non-randomness in the component C: all the possible initial values of the LFSR get tested, and the correct value results in the biased binary stream that matches the non-random output from the component C. The component C is a key-dependent 32-bit-to-32-bit S-box. Vaudenay [124], Murphy

33

and Robshaw [97] have stated that the key-dependent S-boxes may be weak. Berbain and Gilbert's attack on ABC v1 deals with the weak keys that are related to the non-bijective S-box. This type of weak key exists with probability close to 1. Recovering the internal state of a weak key requires about 2^{95} operations and 2^{34} keystream bytes.

In order to resist these attacks, the ABC designers introduced ABC v2 with the improved components A and B. In ABC v2 [5], the length of the LFSR is 127 bits instead of the 63 bits in ABC v1. The increased LFSR length makes it impossible to test all the states of the LFSR, thus the attack on ABC v1 can no longer be applied to ABC v2. However ABC v2 is still insecure due to the low weight of the LFSR and the non-randomness in the component C (the component C in ABC v1 is the same as in ABC v2).

In this chapter, we find a new type of weak key that exists with probability 2^{-32} . This new type of weak key results in a heavily biased output of the component C. Due to the low weight of the LFSR and the strong correlation resulting from the component C, a fast correlation attack can be applied to recover the LFSR. After recovering the LFSR, the internal state of the cipher can be recovered easily. The identification of a weak key from 2^{32} random keys requires 6460 keystream bytes from each key, and $2^{13.5}$ instructions for each keystream. Recovering the internal state of a weak key requires about $2^{27.5}$ keystream bytes and $2^{35.7}$ instructions. Both the ABC v1 and ABC v2 are vulnerable to this attack.

This chapter is organized as follows. In Sect. 3.2, we illustrate the ABC v2. In Sect. 3.3, we define the weak keys and show how to identify them. Section 3.4 recovers the internal state of a weak key. Section 3.5 concludes this chapter.

3.2 The Stream Cipher ABC v2

The stream cipher ABC v2 consists of three components – A, B and C, as shown in Fig. 3.1 [5]. The component A is a regularly clocked LFSR, the component B is a finite state machine (FSM), and the component C is a key-dependent S-box. ABC v1 has the same structure as ABC v2 except that the LFSR in ABC v1 is 63-bit, and the FSM in ABC v1 has less elements than that in ABC v2. The component C in ABC v1 is the same as that in ABC v2.

The component A is based on a linear feedback shift register with primitive polynomial $g(x) = x^{127} + x^{63} + 1$. Denote the register in component A as $(\overline{z}_3, \overline{z}_2, \overline{z}_1, \overline{z}_0)$, where each \overline{z}_i is a 32-bit number. Note that this 128-bit register itself is not a linear feedback shift register. Its initial value depends on the key



Figure 3.1: Keystream generation of ABC v2

and IV. At each step of ABC v2, 32 bits of this 128-bit register get updated as

$$\zeta = (\overline{z}_2 \oplus (\overline{z}_1 \ll 31) \oplus (\overline{z}_0 \gg 1)) \mod 2^{32}$$
$$\overline{z}_0 = \overline{z}_1, \ \overline{z}_1 = \overline{z}_2, \ \overline{z}_2 = \overline{z}_3, \ \overline{z}_3 = \zeta,$$

where \ll and \gg indicates left shift and right shift, respectively.

The component B is specified as $B(x) = ((x \oplus d_0) + d_1) \oplus d_2 \mod 2^{32}$, where x is the 32-bit input, d_0 , d_1 and d_2 are key and IV dependent 32-bit numbers, $d_0 \equiv 0 \mod 4$, $d_1 \equiv 1 \mod 4$, $d_2 \equiv 0 \mod 4$. The x is updated as $x = B(x) + \overline{z}_3$.

The component C is specified as $C(x) = S(x) \gg 16$, where \gg indicates rotation, x is the 32-bit input, $S(x) = e + \sum_{i=0}^{31} (e_i \times x[i])$, where x[i] denotes the *i*th least significant bit of x, and e and e_i are key dependent 32-bit random numbers, except that $e_{31} \equiv 2^{16} \mod 2^{17}$. Note that e and e_i are not related to the initialization vector.

Each 32-bit keystream word is given as $y = C(x) + \overline{z}_0$.

The details of the initialization of ABC v2 are not described here. We are only interested in the generation of the key-dependent S-box in the component C. The above specifications of the component C are sufficient for the illustration of the attacks presented in this chapter.

3.3 The Weak Keys of ABC v2

In Sect. 3.3.1, we introduce some observation related to the bias of carry bits. Section 3.3.2 defines the ABC v2 weak keys and gives an attack to identify them.

3.3.1 The large bias of carry bits

Carry bits are always biased even if the two addends are random. The probability that the value of the carry bit at the *n*-th least significant bit position is 0 is $\frac{1}{2} + \frac{1}{2^{n+1}}$ $(n \ge 1)$. However, this bias is very small for large *n*. In the following, we look for the large bias of carry bits when the addends are not random. Lemma 2.1 implies the following bias:

Theorem 3.1. Denote a_i , b_i $(1 \le i \le 3)$ as *n*-bit integers. Denote c_i $(1 \le i \le 3)$ as binary values satisfying $c_i = (a_i + b_i) \gg n$. Let a_1, a_2, b_1, b_2 and b_3 be random and independent, but $a_3 = a_1 \oplus a_2$. Then $c_1 \oplus c_2 \oplus c_3$ is biased. For n = 16, $\Pr(c_1 \oplus c_2 \oplus c_3 = 0) \approx 0.5714$.

If we apply Lemma 2.1 directly, we obtain that $Pr(c_1 \oplus c_2 \oplus c_3 = 0) = \frac{1}{2} + \frac{1}{16} = 0.5625$. (The u_{n-1} 's in Lemma 2.1 are eliminated since they are linearly related in Theorem 3.1.) The small difference between these two biases (0.5714 and 0.5625) is because that a_3 is not an independent random number.

We illustrate the validity of Theorem 3.1 with numerical analysis. For small n, we try all the values of a_1 , a_2 , b_1 , b_2 and b_3 and the results are given in Table 3.1. From Table 3.1, we see that the bias ϵ converges to 0.0714 as the value of n increases. For n = 16, we performed 2^{32} tests, and the bias is about 0.071424. For n = 32, the bias is about 0.071434 with 2^{32} tests. The experimental results show that Theorem 3.1 is valid. Recently, the complete proof of Theorem 3.1 is given in [137]. It was shown that $\Pr(c_1 \oplus c_2 \oplus c_3 = 0) = \frac{4}{7} + \frac{3}{7} \times \frac{1}{8^n}$, which confirms the correctness of Theorem 3.1.

Remarks. In Theorem 3.1, if a_1 , a_2 , a_3 , b_1 , b_2 and b_3 are all random and independent, then $\Pr(c_1 \oplus c_2 \oplus c_3 = 0) = \frac{1}{2} + 2^{-3n-1}$, which is very small for n = 16. This small bias cannot be exploited to break ABC v2.

Table 3.1: The probability of $c_1 \oplus c_2 \oplus c_3 = 0$ (denote the probability as $\frac{1}{2} + \epsilon$)

| n | ϵ | n | ϵ |
|---|----------------|---|----------------------------|
| 1 | 0.125 | 5 | 0.071441650390625 |
| 2 | 0.078125 | 6 | 0.071430206298828125 |
| 3 | 0.072265625 | 7 | 0.071428775787353515625 |
| 4 | 0.071533203125 | 8 | 0.071428596973419189453125 |

3.3.2 Identifying the weak keys

We start the attack with analyzing the linear feedback shift register used in ABC v2. The register $(\overline{z}_3, \overline{z}_2, \overline{z}_1, \overline{z}_0)$ is updated according to the primitive polynomial $g(x) = x^{127} + x^{63} + 1$. Note that each time the 127-bit LFSR advances 32 steps. To find a linear relation of the 32-bit words, we take the 2⁵th power of g(x), and obtain

$$g^{2^{5}}(x) = x^{127 \times 32} + x^{63 \times 32} + 1.$$
(3.1)

Denote the z_0 at the *i*-th step as z_0^i , and denote the *j*th significant bit of z_0^i as $z_{0,j}^i$. Since each time 32 bits get updated, the distance between $\overline{z}_{0,j}^i$ and $\overline{z}_{0,j}^{i+k}$ is $|32 \cdot (k-i)|$. According to (3.1), we obtain the following linear recurrence

$$\overline{z}_0^i \oplus \overline{z}_0^{i+63} \oplus \overline{z}_0^{i+127} = 0.$$
(3.2)

The weak keys of ABC v2 are related to the S(x) in the component C. S(x) is defined as $S(x) = e + \sum_{i=0}^{31} (e_i \times x[i])$, where e and e_i are key dependent 32bit random numbers, except that $e_{31} \equiv 2^{16} \mod 2^{17}$. If the least significant bits of e and e_i $(0 \le i < 32)$ are all 0, then the least significant bit of S(x) is always 0, and we consider the key as weak key. Note that the least significant bit of e_{31} is always 0. Thus a randomly chosen key is weak with probability 2^{-32} .

In the following, we describe how to identify the weak keys. Denote the 32-bit keystream word at the *i*th step as y_i , the *j*th significant bit of y_i as $y_{i,j}$. And denote x_i as the input to function S at the *i*-th step. Then $y_i = (S(x_i) \gg 16) + \overline{z}_0^i$. Let $c_{i,j}$ denote the carry bit at the *j*-th least significant bit position of $(S(x_i) \gg 16) + \overline{z}_0^i$, i.e., $c_{i,j} = (((S(x_i) \gg 16) \mod 2^j) + (\overline{z}_0^i \mod 2^j)) \gg j$. Assume that $((S(x_i) \gg 16) \mod 2^{16}$ is random. According to Theorem 3.1 and (3.2), we obtain

$$\Pr(c_{i,16} \oplus c_{i+63,16} \oplus c_{i+127,16} = 0) = \frac{1}{2} + 0.0714.$$
(3.3)

Due to the rotation of $S(x_i)$, we know that

$$y_{i,16} = S(x_i)_0 \oplus z_{0,16}^i \oplus c_{i,16} , \qquad (3.4)$$

where $S(x_i)_0$ denotes the least significant bit of $S(x_i)$. Note that $S(x_i)_0$ is always 0 for a weak key. From (3.2) and (3.4), we obtain

$$y_{i,16} \oplus y_{i+63,16} \oplus y_{i+127,16} = c_{i,16} \oplus c_{i+63,16} \oplus c_{i+127,16}.$$
(3.5)

From (3.3) and (3.5), $y_{i,16}$ is biased as

$$\Pr(y_{i,16} \oplus y_{i+63,16} \oplus y_{i+127,16} = 0) = \frac{1}{2} + 0.0714.$$
(3.6)

We use (3.6) to identify the weak keys. Approximate the binomial distribution with the normal distribution. Denote the total number of samples as N, the mean as μ , and the standard deviation as σ . For the binomial distribution, $p = \frac{1}{2}$, $\mu = Np$ and $\sigma = \sqrt{Np(1-p)}$. For (3.6), $p' = \frac{1}{2} + 0.0714$, $\mu' = Np'$ and $\sigma' = \sqrt{Np'(1-p')}$. For the normal distribution, the cumulative distribution function gives value $1 - 2^{-39.5}$ at 7σ , and value 0.023 at -2σ . If the following relation holds

$$u' - u \ge 7\sigma + 2\sigma', \tag{3.7}$$

then in average, each strong key is wrongly identified as weak key (false positive) with probability $2^{-39.5}$, and each weak key is not identified as weak key (false negative) with probability 0.023. It means that the weak keys can be successfully identified since one weak key exists among 2^{32} keys. Solving (3.7), the number of samples required is N = 3954. For each sample, we only need to perform two XORs and one addition. With 3594+127 = 4081 outputs from each key, we can successfully identify the weak keys of ABC v2.

The number of outputs can be reduced if we consider the 2^i th power of g(x) for i = 5, 6, 7, 8. With 1615 outputs, we can obtain 3956 samples. Thus the keystream required in the identification of the weak keys can be reduced to 1615 outputs.

The identification of a weak key implies directly a distinguishing attack on ABC v2. If there are 2^{32} keystreams generated from 2^{32} random keys, and each key produces 1615 outputs, then the keystream can be distinguished from random with high probability. In order to find one weak key, the total number of keystream required is $2^{32} \times 1615 \times 4 = 2^{44.7}$ bytes, and the amount of computation required is $2^{32} \times 3956 \times 2 \approx 2^{45}$ XORs and 2^{44} additions.

Experiment 3.1. We use the original ABC v2 source code provided by the ABC v2 designers in the experiment. After testing 2^{34} random keys, we obtain

38

five weak keys, and one of them is (fe 39 b5 c7 e6 69 5b 44 00 00 00 00 00 00 00 00 00 00). From this weak key we generate 2^{30} outputs, and the bias defined in (3.6) is 0.5714573. The experimental results confirm that the probability that a randomly chosen key is weak is about 2^{-32} , and the bias of a weak key keystream is large.

3.4 Recovering the Internal State

Once a weak key is identified, we proceed to recover the internal state resulting from the weak key. In Sect. 4.1, we apply the fast correlation attack to recover the LFSR. The components B and C are recovered in Sect. 4.2. The complexity of the attack is given in Sect. 3.4.3. Section 3.4.4 applies the attack to ABC v1.

3.4.1 Recovering the initial value of the LFSR

The initial value of the LFSR is recovered by exploiting the strong correlation between the LFSR and the keystream. From Lemma 3.1, we get

$$\Pr(\overline{z}_{0,15}^{i} \oplus c_{i,16} = 0) = \frac{3}{4}.$$
(3.8)

From (3.8) and (3.4), we obtain the following correlation:

$$\Pr(\overline{z}_{0,16}^{i} \oplus \overline{z}_{0,15}^{i} \oplus y_{i,16} = 0) = \frac{3}{4}.$$
(3.9)

The strong correlation in (3.9) indicates that the cipher is very weak.

The fast correlation attack Algorithm A of Meier and Staffelbach [89] can be applied to recover the LFSR. There are some advanced fast correlation attacks [93, 33, 75], but the original attack given by Meier and Staffelbach is sufficient here since we are dealing with a strong correlation and a two-tap LFSR.

We now apply the fast correlation attack Algorithm A [89] to recover the LFSR. Let $p = \frac{3}{4}$, $u_i = \overline{z}_{0,16}^i \oplus \overline{z}_{0,15}^i$, and $w_i = y_{i,16}$. By squaring the polynomial (3.1) iteratively, we obtain a number of linear relations for every u_i :

$$u_i \oplus u_{i+63\cdot 2^j} \oplus u_{i+127\cdot 2^j} = 0 \quad (j \ge 0).$$
 (3.10)

From (3.9) and (3.10), we obtain

$$s = \Pr(w_i \oplus w_{i+63 \cdot 2^j} \oplus w_{i+127 \cdot 2^j} = 0 \mid u_i = w_i) = p^2 + (1-p)^2, \qquad (3.11)$$

where each value of j indicates one relation for w_i (also for $w_{i+63\cdot 2^j}$ and $w_{i+127\cdot 2^j}$). In average there are m relations for w_i as

$$m = m(N, k, t) = (t+1) \cdot \log_2(\frac{N}{2k}),$$
 (3.12)

where N is the number of outputs, k = 127 (the length of the LFSR), t = 2 (taps) for ABC v2. The probability that w_i satisfies at least h of the m relations equals

$$Q(h) = \sum_{i=h}^{m} \binom{m}{i} \cdot (p \cdot s^{i} \cdot (1-s)^{m-i} + (1-p) \cdot (1-s)^{i} \cdot s^{m-i}).$$
(3.13)

If $u_i = w_i$, then the probability that w_i satisfies h of these m relations is equal to

$$R(h) = \sum_{i=h}^{m} \binom{m}{i} \cdot p \cdot s^{h} \cdot (1-s)^{m-h} .$$
 (3.14)

According to [89], $N \cdot Q(h)$ is the number of u_i 's being predicted in the attack, and $N \cdot R(h)$ is the number of u_i 's being correctly predicted.

For N = 4500, there are in average about 12 relations for each w_i . For h = 11, 98.50 bits can be predicted with 98.32 bits being predicted correctly. For h = 10, 384.99 bits can be predicted with 383.21 bits being predicted correctly. To predict 127 bits, we can predict 98.50 bits for h = 11, then predict 127-98.50 = 28.50 bits using the w_i 's satisfying only 10 relations. Then in average there are $98.32 + 28.50 \times \frac{383.21 - 98.31}{384.99 - 98.50} = 126.66$ bits being predicted correctly. It shows that 127 u_i 's can be determined with about 0.34 wrong bits. Then the LFSR can be recovered by solving 127 linear equations.

We carry out an experiment to verify the above analysis. In order to reduce the programming complexity, we consider only the w_i 's with 12 relations, thus we use 8000 outputs in the experiment. Using more outputs to recover the LFSR has no effect on the overall attack since recovering the component B requires about $2^{17.5}$ outputs, as shown in Sect. 4.2.

Experiment 3.2. From the weak key (fe 39 b5 c7 e6 69 5b 44 00 00 00 00 00 00 00 00 00), we generate 8000 outputs, but consider only those $8000 - 2 \cdot 2^{\frac{12}{3}} \cdot 127 = 3936 w_i$'s with 12 relations. We repeat the experiments 256 times with different IVs. For h = 11, 104.66 bits can be predicted with 104.35 bits being predicted correctly. For the w_i 's satisfying only 10 relations, 278.37 bits can be predicted with 276.08 bits being predicted correctly. To predict 127 bits, $127 - 104.66 = 22.34 w_i$'s satisfying only 10 relations should be used. Among the 127 predicted bits, $104.35 + 22.34 \times \frac{276.08}{278.37} = 126.51$ bits are correct.

3.4.2 Recovering the components B and C

After recovering the LFSR, we proceed to recover the component B. In the previous attack on ABC v1 [14], about 2^{77} operations are required to recover the components B and C. That complexity is too high. We give here a very simple method to recover the components B and C with about $2^{33.3}$ operations.

In ABC v2, there are four secret terms in the component B: x, d_0 , d_1 , and d_2 , where d_0 , d_1 and d_2 are static, x is updated as

$$x_i = (((x_{i-1} \oplus d_0) + d_1) \oplus d_2) + \overline{z}_3^i \mod 2^{32}.$$
(3.15)

Note that the more significant bits never affect the less significant bits. It allows us to recover x, d_0 , d_1 , and d_2 bit-by-bit.

Since the initial value of the LFSR is known, the value of each \overline{z}_0^i can be computed, thus we know the value of each $S(x_i)$. On average, the probability that $x_i = x_j$ is about 2^{-32} . For a weak key, the least significant bit of $S(x_i)$ is always 0, and the probability that $S(x_i) = S(x_j)$ is about $2^{-32} + (1-2^{-32}) \cdot 2^{-31} \approx 2^{-32} + 2^{-31}$. Given $2^{17.5}$ outputs, there are about $\binom{2^{17.5}}{2} \times (2^{-32} + 2^{-31}) \approx 12$ cases when $S(x_i) = S(x_j)$ $(i \neq j)$. And there are about $\binom{2^{17.5}}{2} \times 2^{-32} \approx 4$ cases that $x_i = x_j$ among those 12 cases. Choose four cases from those 12 cases randomly, the probability that $x_{i_u} = x_{j_u}$ for $0 \leq u < 4$ is $(\frac{4}{12})^4 = \frac{1}{81}$ (here (i_u, j_u) indicates one of those 12 pairs (i, j) satisfying $S(x_i) = S(x_j)$ $(i \neq j)$).

The value of each \overline{z}_{3}^{i} in (3.15) is already known. When we solve the four equations $x_{i_{u}} = x_{j_{u}}$ ($0 \le u < 4$) to recover x, d_{0}, d_{1} , and d_{2} , we obtain the four unknown terms bit-by-bit from the least significant bit to the most significant bit. The four most significant bits cannot be determined exactly, but the four least significant bits can be determined exactly since only the least significant bit of x is unknown. (We mention here during this bit-by-bit approach, the four bits at each bit position may not be determined exactly, and further filtering is required in such computations.) On average, we expect that solving each set of four equations gives about 8 possible values of x, d_{0}, d_{1} , and d_{2} . Also note that each set of four equations holds true with probability $\frac{1}{81}$, we have about $81 \times 8 = 648$ possible solutions for x, d_{0}, d_{1} , and d_{2} .

After recovering the component B, we know the input and output of each $S(x_i)$, so the component C can be recovered by solving 32 linear equations. This process is repeated 648 times since there are about 648 possible solutions of x, d_0 , d_1 , and d_2 . The exact B and C can be determined by generating some outputs and comparing them to the keystream.

3.4.3 The complexity of the attack

According to the experiment, recovering the LFSR requires about 8000 outputs. For each w_i , testing 12 relations requires about $\frac{12 \cdot 2}{3} = 8$ XORs and 12 additions. After predicting 127 u_i 's, each u_i should be expressed in terms of the initial state of the LFSR. It is almost equivalent to running the LFSR 8000 \cdot 32 steps, with the LFSR being initialized with only one non-zero bit $\overline{z}_{1,31}^0$. Advancing the LFSR 32 steps requires 2 XORs and 2 shifts. Solving a set of 127 binary linear equations requires about $\frac{2 \cdot 127^3}{3} \cdot \frac{1}{32} \approx 42675$ operations on the 32-bit microprocessor. So about $2^{17.8}$ operations are required to recover the LFSR.

Recovering the component B requires about $2^{17.5}$ outputs and solving 81 sets of equations. Each set of equations can be solved bit-by-bit, and it requires about $32 \cdot 2^4 \cdot 2^{17.5} = 2^{26.5}$ operations. Recovering the component C requires solving 648 sets of equations. Each set of equations consists of 32 linear equations with binary coefficients, and solving it is almost equivalent to inverting a 32×32 binary matrix which requires about $\frac{2 \cdot 32^3}{3} \cdot \frac{1}{32} \approx 683$ operations. So $81 \cdot 2^{26.5} + 648 \cdot 683 = 2^{32.8}$ operations are required to recover the components B and C.

Recovering the internal state of a weak key requires $2^{17.5}$ outputs and $2^{17.8} + 2^{32.8} \approx 2^{32.8}$ operations in total.

3.4.4 The attack on ABC v1

The previous attack on ABC v1 deals with a general type of weak keys [14], but the complexity is too high $(2^{95} \text{ operations})$. The above attack can be slighty modified and applied to break ABC v1 (with the weak keys defined in Sect. 3.3) with much lower complexity. We outline the attack on ABC v1 below.

The LFSR in ABC v1 is 63 bits. The shorter LFSR results in more relations for the same amount of keystream. Identifying a weak key requires 1465 outputs from each key instead of the 1615 outputs required in the attack on ABC v2. In theory, recovering the LFSR with the fast correlation attack requires 2500 outputs instead of the 4500 outputs required in the attack on ABC v2. The component B in ABC v1 has only three secret variables. Recovering the component B requires $2^{17.3}$ outputs, with the complexity reduced to $2^{30.1}$ operations, smaller than the $2^{32.8}$ operations required to recover the component B of ABC v2. In total the attack to recover the internal state of ABC v1 with a weak key requires $2^{17.3}$ outputs and $2^{30.1}$ operations.

3.5 Conclusion

Due to the large number of weak keys and the serious impact of each weak key, ABC v1 and ABC v2 are practically insecure.

In order to resist the attack presented in this paper, a straightforward solution is to ensure that at least one of the least significant bits of the 33 elements in the component B should be nonzero. However, ABC v2 with such improvement is still insecure. A new type of weak keys is that at two adjacent bit positions (except the least significant bit position), all the bits of the 33 elements in the component B are 0. After eliminating all the similar weak keys, the linear relation in (3.2) can still be applied to exploit the non-randomness in the outputs of the component C to launch a distinguishing attack. ABC v3 is the latest version of ABC, and it eliminates the weak keys described in this chapter. However, a recent attack exploiting the non-randomness in the outputs of the component C is still able to identify a new weak key with about 2^{60} outputs [136]. It seems difficult to improve the ABC cipher due to the risky design that the 32-bit-to-32-bit S-box is generated from only 33 key-dependent elements.

We recommend updating the secret S-box of ABC v2 frequently during the keystream generation process. In ABC v2, the key-dependent S-box is fixed. For a block cipher design, the S-box has to remain unchanged, but such restriction is not applicable to a stream cipher. Suppose that the size of the key-dependent S-box of a stream cipher is large (it is risky to use the small randomly generated key-dependent S-box). We can update the S-box frequently, such as updating at least one element of the S-Box at each step (in a cyclic way to ensure that all the elements of the S-box get updated) with enough random information in an unpredictable way. When a weak S-box appears, only a small number of outputs are generated from it before the weak S-box disappears, and it becomes extremely difficult for an attacker to collect enough outputs to analyze a weak S-box. Thus an attacker has to deal with the average property of the S-box, instead of dealing with the weakest S-box. For example, the eSTREAM submissions HC-256 [128], HC-128 [135], Py [25] and Pypy [26] use the frequently updated large S-boxes to reduce the effect resulting from the weak S-boxes. The security of ABC stream cipher can be improved in this way, but its performance will be affected.

Chapter 4

Resynchronization Attack I Linear Attack on DECIM

Abstract. DECIM is a hardware oriented stream cipher with an 80-bit key and a 64-bit IV. In this chapter, we point out two serious flaws in DECIM. One flaw is in the initialization of DECIM. It allows to recover about half of the key bits bitby-bit when one key is used with about 2^{20} random IVs; only the first two bytes of each keystream are needed in the attack. The amount of computation required in the attack is negligible. Another flaw is in the keystream generation algorithm of DECIM. The keystream is heavily biased: any two adjacent keystream bits are equal with probability about $\frac{1}{2} + 2^{-9}$. A message could be recovered from the ciphertext if that message is encrypted by DECIM for about 2^{18} times. DECIM with an 80-bit key and an 80-bit IV is also vulnerable to these attacks.

4.1 Introduction

DECIM [11] is a stream cipher submitted by Berbain, Billet, Canteaut, et al. to the ECRYPT stream cipher project [45]. The main feature of DECIM is the use of the ABSG decimation mechanism [11], an idea similar to the shrinking generator [34, 90]. Another excellent feature is that a 32-bit buffer is used in DECIM to ensure that at each step DECIM generates one output bit.

In this chapter, we point out two flaws in DECIM, one in the initialization algorithm, and another one in the keystream generation algorithm. The flaw in the initialization allows for any easy key recovery from the keystreams when one key is used with about 2^{20} random IVs. The flaw in the keystream generation algorithm results in a heavy bias in the keystream, hence the cipher is vulnerable to a broadcast attack.

45

In Sect. 4.2 we describe the DECIM cipher. Section 4.3 presents a key recovery attack on DECIM. The key recovery attack on DECIM is improved in Sect. 4.4. The broadcast attack on DECIM is described in Sect. 4.5. Section 4.6 shows that DECIM with an 80-bit IV is also vulnerable to the attacks. Section 4.7 concludes this chapter.

4.2 Stream Cipher DECIM

DECIM uses the ABSG decimation mechanism in the keystream generation in order to achieve high security and design simplicity. The keystream generation process and the key/IV setup are illustrated in Sect. 4.2.1 and 4.2.2, respectively.

4.2.1 Keystream Generation

The keystream generation diagram of DECIM is given in Fig. 4.1 [11]. DECIM has a regularly clocked LFSR which is defined by the feedback polynomial

$$P(X) = X^{192} + X^{189} + X^{188} + X^{169} + X^{156} + X^{155} + X^{132} + X^{131} + X^{94} + X^{77} + X^{46} + X^{17} + X^{16} + X^5 + 1$$

over GF(2). The related recursion is given as

$$\begin{array}{lll} s_{192+n} & = & s_{187+n} \oplus s_{176+n} \oplus s_{175+n} \oplus s_{146+n} \oplus s_{115+n} \oplus s_{98+n} \oplus s_{61+n} \\ & & \oplus s_{60+n} \oplus s_{37+n} \oplus s_{36+n} \oplus s_{23+n} \oplus s_{4+n} \oplus s_{3+n} \oplus s_n \,. \end{array}$$

At each stage, two bits are generated from the LFSR as follows:

$$y_{t,1} = f(s_{t+1}, s_{t+32}, s_{t+40}, s_{t+101}, s_{t+164}, s_{t+178}, s_{t+187}),$$

$$y_{t,2} = f(s_{t+6}, s_{t+8}, s_{t+60}, s_{t+116}, s_{t+145}, s_{t+181}, s_{t+191}),$$

where the Boolean function f is defined as

$$f(x_{i_1}, ..., x_{i_7}) = \sum_{1 \le j < k \le 7} x_{i_j} x_{i_k}$$

The binary sequence y consists of all the $y_{t,1}$ and $y_{t,2}$ as

$$y = y_{0,1}y_{0,2}y_{1,1}y_{1,2}\cdots y_{t,1}y_{t,2}\cdots$$

The keystream sequence z is generated from the binary sequence y through the ABSG decimation algorithm. The sequence y is split into subsequences of the form (\bar{b}, b^i, \bar{b}) , with $i \ge 0$ and $b \in \{0, 1\}$; \bar{b} denotes the complement of b in $\{0, 1\}$.

For every subsequence (\bar{b}, b^i, \bar{b}) , the output bit is b for i = 0, and \bar{b} otherwise. The ABSG algorithm is given below

> Input: $(y_0, y_1, ...)$ Set: $i \leftarrow 0; j \leftarrow 0;$ Repeat the following steps:

> > $e \leftarrow y_i, z_j \leftarrow y_{i+1}, i \leftarrow i+1;$ while $(y_i = \bar{e}) i \leftarrow i+1;$ $i \leftarrow i+1;$ output $z_j; j \leftarrow j+1;$



Figure 4.1: Keystream Generation Diagram of DECIM

Remarks. The above description of the ABSG and the pseudo-code of ABSG are quoted from [11]. However the outputs of the pseudo-code are the complements of that of the ABSG algorithm. Anyway, this difference has no effect on the security of DECIM. In the rest of the chapter, we assume that the DECIM uses the pseudo-code of ABSG given above.

DECIM is designed to output one bit every two stages. A 32-bit buffer is used to ensure that the probability that there is no output bit is extremely small (2^{-89}) .

4.2.2 Initialization

The secret key K is an 80-bit key. The 64-bit IV is expanded to an 80-bit vector by adding zeros from position 64 up to position 79. The initial value of the LFSR state is loaded as follows

$$s_{i} = \begin{cases} K_{i} \lor IV_{i} & \text{for } 0 \le i \le 55 \\ K_{i-56} \land \overline{IV}_{i-56} & \text{for } 56 \le i \le 111 \\ K_{i-112} \oplus IV_{i-112} & \text{for } 112 \le i \le 191 \end{cases}$$

The LFSR is clocked 192 times. After the *t*-th clocking, $y_{t,1}$ and $y_{t,2}$ are XORed to $x_{t,192}$ as

 $s_{t+192} = s_{t+192} \oplus y_{t,1} \oplus y_{t,2}$.

Then one of two permutations π_1 and π_2 is applied to permute 7 elements s_{t+5} , s_{t+31} , s_{t+59} , s_{t+100} , s_{t+144} , s_{t+177} , s_{t+186} . Two bits $y_{t,1}$ and $y_{t,2}$ are input to the ABSG, if the output of the ABSG is 1, then π_1 is applied; if the output of the ABSG is 0 or if there is no output, then π_2 is applied. The two permutations are defined as

$$\pi_1 = (1 \ 6 \ 3)(4 \ 5 \ 2 \ 7), \pi_2 = (1 \ 4 \ 7 \ 3 \ 5 \ 2 \ 6).$$

4.3 Key Recovery Attack on DECIM

In this section, we develop attacks to recover the secret key of DECIM. This non-optimized attack applies when the same secret key is used with a number of random IVs, and the first 3 bytes of each keystream are known. The optimized attack is given in the next section.

4.3.1 The effects of the permutations π_1 and π_2

The two permutations in the initialization stage of DECIM provide high nonlinearity to the initialization process. However, the permutations also cause some bits in the LFSR to be updated in an improper way. This has a very negative impact on the security of DECIM.

The permutation π_1 is poorly designed. In order to investigate the effects of this permutation, we analyze a weak version by assuming that only this permutation is used in the initialization process, i.e., we replace π_2 with π_1 . The values of 140 elements in the LFSR $(s_5, s_6, \ldots, s_{58}, \text{ and } s_{100}, s_{101}, \ldots, s_{185})$ would never be updated by the initialization process. For example, s_{21} would always become s_{192+6} . The details are given below. We trace the bit s_{21} , after 16 steps it becomes s_{16+5} due to the shift of the LFSR. Then it becomes s_{16+177} due to the permutation π_1 . After 33 steps, it becomes s_{49+144} due to the shift of the LFSR. Then it becomes s_{49+31} due to the permutation π_1 . After 26 steps, it becomes s_{75+5} due to the shift of the LFSR. Then it becomes s_{75+177} due to the permutation π_1 . This process repeats and at the end of the initialization process, it becomes s_{192+6} .

The first bit of the keystream is given as $y_{192,2}$; it is computed as $y_{192,2} = f(s_{192+6}, s_{192+8}, s_{192+60}, s_{192+116}, s_{192+145}, s_{192+181}, s_{192+191})$. By tracing the bits of the LFSR during the initialization process, we know that $s_{192+6} \leftarrow s_{21}$, $s_{192+8} \leftarrow s_{23}, s_{192+116} \leftarrow s_{132}, s_{192+145} \leftarrow s_{160}, s_{192+181} \leftarrow s_{33}$. If every key and IV pair is randomly generated, then according to the loading of the key and IV, we know that s_{21}, s_{23} , and s_{33} take value 1 with probability 0.75. Thus according to the definition of the function f, the value of $y_{192,2}$ is 0 with probability 0.582. So the first bit of the keystream is heavily biased. It shows that the effect of the permutation π_1 is terrible.

In DECIM, there are two permutations, π_1 and π_2 . They are chosen according to the output of ABSG: π_1 is chosen with probability $\frac{1}{3}$, π_2 with probability $\frac{2}{3}$. Due to these two permutations, the average number of bits that are not updated by the initialization process is reduced to 54.5 (obtained by running 2¹⁶ random key and IV pairs). It shows that the permutations π_1 and π_2 which are chosen by the output of ABSG have a negative impact on the security of DECIM.

4.3.2 Recovering K_{21}

In the initialization process, we monitor the bit s_{21} . s_{21} becomes s_{192+6} with probability $\frac{1}{27}$. If s_{192+6} takes the value 0, and all the other bits in the LFSR at the 192-th step are distributed uniformly, then the value of the first bit of the keystream is 0 with probability $q_0 = \frac{56}{128}$. If s_{192+6} takes the value 1, and all the other bits of the LFSR at the 192-th step are distributed uniformly, then the value of the first bit of the keystream is 0 with probability $q_1 = \frac{72}{128}$. Denote the probability that the value of the first keystream bit is 0 when $s_{21} = 0$ as p_0 , and the probability that the value of the first keystream bit is 0 when $s_{21} = 1$ as p_1 . Then $\Delta p = p_1 - p_0 = \frac{1}{27} \times (q_1 - q_0) = 2^{-7.75}$. In an experiment we chose 2^{20} random IVs for $s_{21} = 0$, and another 2^{20} random IVs for $s_{21} = 1$, and we found that $\Delta p = 2^{-7.99}$. The experimental result confirms that the theoretically predicted result $\Delta p = 2^{-7.75}$ is close to the correct value.

The above property can be applied to recover K_{21} as follows. Suppose that the same key is used with N random IVs to generate keystreams. For the keystreams with $IV_{21} = 0$, we compute the probability that the value of the first bit is 0, and denote this probability as p'_0 . For the keystreams with $IV_{21} = 1$, we compute the probability that the value of the first bit is 0, and denote this probability that the value of the first bit is 0, and denote this probability as p'_1 . If $p'_1 > p'_0$, we decide that $K_{21} = 0$; otherwise, $K_{21} = 1$. For $N = (\frac{\Delta p}{2})^{-2} \times 2 = 2^{18.5}$, the attack can determine the value of K_{21} with success rate 0.977.

4.3.3 Recovering $K_{22}K_{23}\ldots K_{30}$

By tracing the bits in the initialization process, we notice that each s_{22+i} is mapped to $s_{192+7+i}$ with probability $\frac{1}{27}$ for $0 \le i \le 8$ (each of them is only mapped by π_1 at s_{t+5}). We know that $s_{22+i} = K_{22+i} \lor IV_{22+i}$, and $s_{192+7+i}$, $s_{192+9+i}$ are used in the generation of $y_{193+i,2}$ for $0 \le i \le 10$. In this section, we show that the key bits $K_{22}K_{23}K_{24}\ldots K_{30}$ can be recovered from the keystream.

An attack similar to that given in Sect. 4.3.2 can be applied to recover the value of K_{23} from the first keystream bits generated from $2^{18.5}$ IVs.

In order to determine the values of K_{22} and K_{24} , we observe the second bit of the keystream. Due to the disturbance of the ABSG, $y_{193,2}$ becomes the second keystream bit with probability 0.5. Thus $\Delta p' = 0.5 \times \Delta p = 2^{-8.75}$. To recover K_{22} and K_{24} , we need $2^{20.5}$ IVs in order to obtain a success probability of 0.977.

In order to determine the value of K_{25} , we observe the second and third bits of the keystream. $y_{194,2}$ becomes the second bit of the keystream with probability $\frac{1}{8}$, and becomes the third bit of the keystream with probability $\frac{1}{4}$. Thus $\Delta p'' = \frac{1}{2} \times (\frac{1}{4} + \frac{1}{8}) \times \Delta p = 2^{-10.165}$. To recover K_{25} , we need $2^{22.3}$ IVs in order to obtain a success probability of 0.977.

We omit the details of recovering $K_{26} \cdots K_{29}$. To recover K_{30} , we observe the fifth, sixth and seventh bits of the keystream. $y_{199,2}$ would become one of these three bits with probability $\frac{77}{256}$. Thus $\Delta p''' = \frac{1}{3} \times \frac{77}{256} \times \Delta p = 2^{-11.068}$. To recover K_{29} , we need $2^{23.55}$ IVs in order to obtain the success rate 0.977.

4.3.4 Recovering $K_9K_{10}...K_{19}$

By tracing the bits in the initialization process, we notice that each s_{9+i} is mapped to $s_{192+166+i}$ with probability $\frac{1}{27}$ for $0 \le i \le 10$ (each of them is only mapped by π_1 at s_{t+5}). We know that $s_{9+i} = K_{9+i} \lor IV_{9+i}$, and $s_{192+166+i}$ is used in the generation of $y_{194+i,1}$ for $0 \le i \le 10$. The attacks given in this section are similar to those given above. We only illustrate how to recover K_9 and K_{19} .

In order to determine the value of K_9 , we observe the second bit of the keystream. $y_{194,1}$ becomes the second bit of the keystream with probability $\frac{1}{4}$. Thus $\Delta p^{(4)} = \frac{1}{4} \times \Delta p = 2^{-9.75}$. To recover K_9 , we need $2^{22.5}$ IVs in order to obtain a success probability of 0.977.

In order to determine the value of K_{19} , we observe the 8-th, 9-th and 10-th bits of the keystream. $y_{204,1}$ becomes one of these three bits with probability 0.260. Thus $\Delta p^{(5)} = \frac{1}{3} \times 0.260 \times \Delta p = 2^{-11.28}$. To recover K_{19} , we need $2^{23.98}$ IVs in order to obtain a success probability of 0.977.

4.3.5 Recovering $K_{32}K_{33}...K_{46}$

By tracing the bits in the initialization process, we notice that each s_{144+i} is mapped to $s_{192+16+i}$ with probability $\frac{1}{27}$ for $0 \le i \le 14$ (each of them is only mapped by π_1 at s_{t+5}). We know that $s_{144+i} = K_{32+i} \oplus IV_{32+i}$, and $s_{192+16+i}$ is used in the generation of $y_{200+i,1}$ for $0 \le i \le 14$.

Since for s_{144+i} ($0 \le i \le 14$), the key bits are XORed with the IV bits, the attack is slightly modified. For example, if the probability of 0 in the keystream for $IV_{32} = 0$ is higher than the probability of 0 in the keystream for $IV_{32} = 1$, then we predict that $K_{32} = 0$; otherwise, $K_{32} = 1$. We only illustrate how to recover K_{32} and K_{46} .

In order to determine the value of K_{32} , we observe the sixth, seventh and eighth bits of the keystream. $y_{200,2}$ becomes one of these three bits with probability 0.28027. Thus $\Delta p^{(6)} = \frac{1}{3} \times 0.28027 \times \Delta p = 2^{-11.17}$. To recover K_{32} , we need $2^{23.755}$ IVs in order to obtain a success probability of 0.977.

In order to determine the value of K_{46} , we assume that starting from the fourth bit of the sequence y, each bit becomes the output with probability $\frac{1}{3}$. Then $y_{214,2}$ becomes one of the 12th, 13th, ..., 18th bits of the keystream with probability 0.16637. Thus $\Delta p^{(7)} = \frac{1}{7} \times 0.16637 \times \Delta p = 2^{-13.145}$. To recover K_{29} , we need $2^{26.482}$ IVs in order to obtain a success probability of 0.977.

The attacks given in this section recover 36 bits of the secret key with about 2^{26} random IVs. For each IV, only the first 3 bytes of the keystream are needed in the attack.

4.4 Improving the Key Recovery Attack

In the above attacks, we deal with the bits affected only by π_1 at s_{t+5} during the initialization (the bits affected by π_2 are not considered in the attack). In order to improve the attack, we have used a computer program to trace all the possibilities for each bit s_i ($0 \le i \le 175$) during the initialization process to find out the distribution of that bit at the end of initialization. Then we have searched the optimal attack for that bit. We performed the experiment, and found that 44 key bits can be recovered with less than 2^{20} IVs, and only the first 2 bytes of the keystream are required in the attack. The experiment results are given in Table A.1.

4.5 The Keystream of DECIM Is Heavily Biased

The nonlinear function f in DECIM is extremely simple. However this Boolean function is balanced but not 1-resilient. Unfortunately the ABSG decimation

mechanism and the buffer in the output function fail to eliminate the bias existing in the output of f, hence the keystream is heavily biased.

4.5.1 The keystream is biased

We start with analyzing the function f

$$f(x_{i_1}, ..., x_{i_7}) = \sum_{1 \le j < k \le 7} x_{i_j} x_{i_k}$$

If any bit of the input of f is equal to 1, then f outputs a '1' with probability $\frac{72}{128}$; otherwise it outputs a '1' with probability $\frac{56}{128}$. Thus for $f(x_{i_1}, ..., x_{i_7})$ and $f(x'_{i_1}, ..., x'_{i_7})$, if one bit of one input is always equal to one bit of another input (i.e., $x_{i_a} = x'_{i_b}$ where $0 \le a, b \le 7$), then the outputs related to these two inputs would be equal with probability $(\frac{56}{128})^2 + (\frac{72}{128})^2 = \frac{65}{128}$.

Note that $y_{t,1}$ and $y_{t,2}$ are computed as follows

$$y_{t,1} = f(s_{t+1}, s_{t+32}, s_{t+40}, s_{t+101}, s_{t+164}, s_{t+178}, s_{t+187}),$$

$$y_{t,2} = f(s_{t+6}, s_{t+8}, s_{t+60}, s_{t+116}, s_{t+145}, s_{t+181}, s_{t+191}).$$

Denote $A = \{1, 32, 40, 101, 164, 178, 187\}, B = \{6, 8, 60, 116, 145, 181, 191\}$, and denote each element of A by a_i , and each element of B by b_i $(1 \le i \le 7)$. Then $y_{t,1} = y_{t+a_i-a_j,1}$ and $y_{t,2} = y_{t+b_i-b_j,2}$ with probability $\frac{65}{128}$ for $1 \le i, j \le 7$ and $i \ne j$. And $y_{t+b_i-a_j,1} = y_{t,2}$ with probability $\frac{65}{128}$ for $1 \le i, j \le 7$. It shows that the binary sequence y is heavily biased.

The heavily biased sequence y is used as input to the ABSG decimation algorithm. It results in a heavily biased output. In the attack, we are interested in those biases in y that would not be significantly reduced by the ABSG Algorithm. Thus we will analyze the bias of $(y_{t+3,1}, y_{t,2})$, $(y_{t+4,1}, y_{t,2})$ and $(y_{t,2}, y_{t+2,2})$ to find out how they affect the randomness of the output of ABSG.

For example, we analyze the effect of the bias of $(y_{t+3,1}, y_{t,2})$. $y_{t+3,1} = y_{t,2}$ with probability $\frac{65}{128}$. Denote the *i*-th bit of the sequence y by y^i . Thus $y^i = y^{i+5}$ with probability $\frac{129}{256}$. (y^i, y^{i+5}) would affect the bias of the output of the ABSG in two approaches. One approach is that (y^i, y^{i+5}) becomes (z_j, z_{j+2}) with probability $\frac{1}{4}$ (case 1: $y_i = y_{i-1}, y_{i+2} \neq y_{i+1}$ and $y_{i+3} = y_{i+2}$; case 2: $y_i \neq y_{i-1}, y_{i+1} = y_{i-1}$ and $y_{i+3} = y_{i+2}$). Thus for this approach, the bias of (y^i, y^{i+5}) causes $z_j = z_{j+2}$ with probability $\frac{513}{1024}$. Another approach is that if $y_i = y_{i-1}$ and $y_{i+2} = y_{i+1}$, then (y_i, y_{i+4}) becomes (z_j, z_{j+2}) . Note that $y_{i+4} = y_{i-1}$ with probability $\frac{129}{256}$, so $z_j = z_{j+2}$ with probability $\frac{129}{256}$. This approach happens with probability $\frac{1}{4}$. Thus the bias of (y^i, y^{i+5}) causes $z_j = z_{j+2}$ with probability $\frac{513}{1024}$. Combining these two approaches, we know that $z_j = z_{j+2}$ with probability $\frac{513}{1024}$.
We continue analyzing the above example since the output of ABSG decimation algorithm should pass through the buffer before becoming keystream. By analyzing the ABSG decimation algorithm and the buffer, we notice that if (y^i, y^{i+5}) becomes $z_j = z_{j+2}$ after the ABSG decimation algorithm, then it becomes $z'_k = z'_{k+1}$ with probability 0.6135 after passing through the buffer; if (y^i, y^{i+4}) becomes $z_j = z_{j+2}$ after the ABSG decimation algorithm, then it becomes $z'_k = z'_{k+1}$ with probability 0.5189 after passing through the buffer. Thus after passing through the buffer, the two approaches lead to $z'_k = z'_{k+1}$ with probability $\frac{1}{2} + 0.6135 \times \frac{1}{1024} + 0.5189 \times \frac{1}{1024} = \frac{1}{2} + 2^{-9.82}$. A similar analysis can be applied to the biases resulting from $(y_{t+4,1}, y_{t,2})$ and

A similar analysis can be applied to the biases resulting from $(y_{t+4,1}, y_{t,2})$ and $(y_{t,2}, y_{t+2,2})$. The bias of $(y_{t,2}, y_{t+2,2})$ would cause $z'_k = z'_{k+1}$ with probability about $\frac{1}{2} + 2^{-10.84}$, and the bias of $(y_{t+4,1}, y_{t,2})$ would cause $z'_k = z'_{k+1}$ with probability about $\frac{1}{2} + 2^{-11.73}$.

Combining the effects of $(y_{t+3,1}, y_{t,2})$, $(y_{t+4,1}, y_{t,2})$ and $(y_{t,2}, y_{t+2,2})$, the bias of $z'_k = z'_{k+1}$ is about $\frac{1}{2} + 2^{-9.82} + 2^{-10.84} + 2^{-11.73} = \frac{1}{2} + 2^{-9.00}$.

Now we verify the above analysis with an experiment. We have generated about 2^{30} keystream bits from DECIM and found that $z'_k = z'_{k+1}$ is about $\frac{1}{2} + 2^{-8.67}$. The experimental result shows that the theoretical result is close to that obtained from the experiment.

4.5.2 Broadcast attack

Due to the bias in the keystream, part of the message could be recovered from the ciphertexts if the same message is encrypted many times using DECIM with random key and IV pairs. A similar attack has been applied to RC4 by Mantin and Shamir [86].

Suppose that one message bit is encrypted N times, and each keystream bit is 0 with probability $\frac{1}{2} + \Delta p$ with $\Delta p > 0$. Denote the number of '0' in the ciphertext bits by n_0 . If $n_0 > \frac{N}{2}$, we conclude that the message bit is equal to '0'; otherwise, we conclude that the message bit is equal to '1'. For $N = \Delta p^{-2}$, the message bit is recovered with a success probability of 0.977.

Thus if one message is encrypted about 2^{18} times with different keys and IVs, the message could be recovered from the ciphertexts.

4.6 Attacks on DECIM with 80-bit IV

The keystream generation algorithm of DECIM with an 80-bit IV is the same as DECIM with a 64-bit IV. Thus DECIM with an 80-bit IV still generates heavily biased keystream and it is vulnerable to the broadcast attack.

The initialization process of DECIM with an 80-bit IV is slightly different from the 64-bit IV version. The key and IV are loaded into the LFSR as

$$s_i = \begin{cases} 0 & \text{for } 0 \le i \le 31 \\ K_{i-32} \oplus IV_{i-32} & \text{for } 32 \le i \le 111 \\ K_{i-112} & \text{for } 112 \le i \le 191 \end{cases}$$

Similar to the attack given in Sect. 4.4, we have carried out an experiment to compute the IVs required to recover each bit. With 2^{21} IVs, 41 bits of the secret key could be recovered. Only the first 2 bytes of the keystream are required in the attack. The experiment results are given in Table A.2.

4.7 Conclusion

In this chapter, we developed two attacks against the stream cipher DECIM. The key could be recovered easily from the keystream with about 2^{20} random IVs. And the keystream of DECIM is heavily biased. The results indicate that DECIM is very weak.

Recently, the designers of DECIM have proposed DECIM v2 [12]. DECIM v2 is much simpler than DECIM. The initialization of DECIM v2 uses 768 steps of the keystream generation algorithm with the output bit being XORed to the LFSR. The filter is changed and f is one-resilient. DECIM v2 is not vulnerable to the attacks presented in this chapter.

Chapter 5

Resynchronization Attack II Differential Attack on WG

Abstract. WG is a stream cipher submitted to eStream – the ECRYPT stream cipher project. In this chapter, we point out security flaws in the resynchronization of WG. The resynchronization of WG is vulnerable to a differential attack. For WG with 80-bit key and 80-bit IV, 48 bits of the secret key can be recovered with about $2^{31.3}$ chosen IVs . For each chosen IV, only the first four keystream bits are needed in the attack.

5.1 Introduction

For the research on stream ciphers, resynchronization atacks have not been studied as thoroughly as the keystream generation algorithm itself. Ten years ago, Daemen, Govaerts and Vandewalle analyzed the weakness of linear resynchronization mechanism with known output Boolean function [41]. Later Golić and Morgari studied linear resynchronization mechanisms with unknown output function [61]. However almost all the stream ciphers proposed recently use non-linear resynchronization mechanisms, so the previous attacks on linear resynchronization mechanisms could no longer be applied. Recently Armknecht, Lano and Preneel applied algebraic attacks and linear cryptanalysis to the resynchronization mechanism and obtained lower bounds for the nonlinearity required from a secure resynchronization mechanism [6]. In this chapter, we apply the differential attack and slide attack to stream ciphers with non-linear resynchronization. We show that the cryptanalysis techniques used to attack block ciphers are also useful in the analysis of non-linear resynchronization mechanisms. WG [103] is a stream cipher submitted to eStream, the ECRYPT stream cipher project [45]. The keystream generation algorithm of WG is quite strong. The keystream generation of WG is based on the WG transformation which has excellent cryptographic properties [62]. However, the resynchronization mechanism of WG is insecure. The resynchronization mechanism of WG is vulnerable to a differential attack [27]. Breaking WG requires $2^{31.3}$ chosen IVs.

This chapter is organized as follows. WG is introduced in Sect. 5.2. The differential attack on WG is presented in Sect. 5.3 Section 5.4 concludes this chapter.

5.2 Description of WG

WG is a hardware oriented stream cipher with key length up to 128 bits; it supports IV sizes from 32 bits to 128 bits. The main feature of the WG stream cipher is the use of the WG transformation to generate keystream from an LFSR.

Keystream Generation



Figure 5.1: Keystream generation diagram of WG

The keystream generation diagram of WG is given in Fig. 5.1 [103]. WG has a regularly clocked LFSR which is defined by the feedback polynomial

$$p(x) = x^{11} + x^{10} + x^9 + x^6 + x^3 + x + \gamma$$
(5.1)

over $GF(2^{29})$, where $\gamma = \beta^{464730077}$ and β is the primitive root of

$$g(x) = x^{29} + x^{28} + x^{24} + x^{21} + x^{20} + x^{19} + x^{18} + x^{17} + x^{14} + x^{12} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x + 1.$$
(5.2)

Then the non-linear WG transformation, $GF(2^{29}) \rightarrow GF(2)$, is applied to generate the keystream from the LFSR.

Resynchronization (Key/IV setup)

The key/IV setup of WG is given in Fig. 5.2. After the key and IV have been loaded into the LFSR, it is clocked 22 steps. During each of these 22 steps, 29 bits from the middle of the WG transformation are XORed to the feedback of LFSR, as shown in Fig. 5.2 [103].



Figure 5.2: Key/IV setup of WG

One step of the key/IV setup can be expressed as follows:

 $T = S(1) \oplus S(2) \oplus S(5) \oplus S(8) \oplus S(10) \oplus (\gamma \times S(11)) \oplus WG'(S(11)),$ $S(i) = S(i-1) \text{ for } i = 11 \cdots 2; S(1) = T,$

where WG'(S(11)) denotes the 29 bits extracted from the WG transformation, as shown in Fig. 5.2.

The WG cipher supports several key and IV sizes: the key size can be 80 bits, 96 bits, 112 bits and 128 bits and the IV sizes can be 32 bits, 64 bits, 80 bits, 96 bits, 112 bits, and 128 bits. Slightly different resynchronization mechanisms are used for the different IV sizes. The details are given in Sect. 5.3.

5.3 Differential Attacks on WG

The resynchronization of WG can be broken with a chosen IV attack based on differential cryptanalysis. (We remind the readers that the details of the differential attack given in this chapter are slightly different from the standard differential attack on a block cipher, such as the generation of the differential pairs and the filtering of the wrong pairs.) WG with a 32-bit IV size is not vulnerable to the attack given in this section (since no special differential can be introduced into this short IV). In Sect. 3.1 the attack is applied to break WG with an 80-bit key and an 80-bit IV. The attacks on WG with IV sizes larger than 80 bits are given in Sect. 5.3.2. The attack on WG with a 64-bit IV size is given in Sect. 5.3.3.

5.3.1 Attack on WG with an 80-bit key and an 80-bit IV

We investigate the security of the key/IV setup of WG with an 80-bit key and an 80-bit IV. For this version of WG, denote the key as $K = k_1, k_2, k_3, \dots, k_{80}$ and the IV as $IV = IV_1, IV_2, IV_3, \dots, IV_{80}$. They are loaded into the LFSR as follows:

| $S_{1,\dots,16}(1) = k_{1,\dots,16}$ | $S_{17,\ldots,24}(1) = IV_{1,\ldots,8}$ |
|--|---|
| $S_{1,\ldots,8}(2) = k_{17,\ldots,24}$ | $S_{9,\ldots,24}(2) = IV_{9,\ldots,24}$ |
| $S_{1,\dots,16}(3) = k_{25,\dots,40}$ | $S_{17,\dots,24}(3) = IV_{25,\dots,32}$ |
| $S_{1,\ldots,8}(4) = k_{41,\ldots,48}$ | $S_{9,\ldots,24}(4) = IV_{33,\ldots,48}$ |
| $S_{1,\dots,16}(5) = k_{49,\dots,64}$ | $S_{17,\ldots,24}(5) = IV_{49,\ldots,56}$ |
| $S_{1,\dots,8}(6) = k_{65,\dots,72}$ | $S_{9,\ldots,24}(6) = IV_{57,\ldots,72}$ |
| $S_{1,\dots,8}(7) = k_{73,\dots,80}$ | $S_{17,\ldots,24}(7) = IV_{73,\ldots,80}$ |

All the remaining bits of the LFSR are set to zero. Then the LFSR is clocked 22 steps with the middle value from the WG transformation being used in the feedback.

The chosen IV attack on WG goes as follows. For each secret key K, we choose two IVs, IV' and IV'', so that IV' and IV'' are identical in 8 bytes, but differ in two bytes: $IV'_{17,...,24} \neq IV''_{17,...,24}$ and $IV'_{49,...,56} \neq IV''_{49,...,56}$. The differences satisfy $IV'_{17,...,24} \oplus IV''_{17,...,24} = IV'_{49,...,56} \oplus IV''_{49,...,56}$.

Denote the value of S(i) $(1 \le i \le 11)$ at the end of the *j*-th step by $S^{j}(i)$, and denote loading the key/IV as the 0th step. After loading the key and the chosen IV into LFSR, we know that the differences in S(2) and S(5) are the same, i.e., $S'^{0}(2) \oplus S''^{0}(2) = S'^{0}(5) \oplus S''^{0}(5)$. We denote this difference as Δ_{1} , i.e., $\Delta_{1} = S'^{0}(2) \oplus S''^{0}(2) = S'^{0}(5) \oplus S''^{0}(5)$.

We now examine the differential propagation during the 22 steps in the key/IV setup. The complete differential propagation is shown in Table 5.1, where the differences at the *i*-th step indicate the differences at the end of the *i*-th step. The difference $\triangle_2 = (\gamma \times S'^6(11) \oplus WG'(S'^6(11)) \oplus (\gamma \times S''^6(11) \oplus WG'(S''^6(11)) = (\gamma \times S''^0(5) \oplus WG'(S''^0(5)) \oplus (\gamma \times S''^0(5) \oplus WG'(S''^0(5)))$. Similarly, we obtain that $\triangle_3 = (\gamma \times S'^0(2) \oplus WG'(S'^0(2)) \oplus (\gamma \times S''^0(2) \oplus WG'(S''^0(2)).$

From Table 5.1, we notice that at the end of the 22th step, the difference at $S^{22}(10)$ is $\Delta_2 \oplus \Delta_3$. From the above description of Δ_2 and Δ_3 , we know that

$$\Delta_2 \oplus \Delta_3 = ((\gamma \times S'^0(5) \oplus WG'(S'^0(5)) \oplus (\gamma \times S''^0(5) \oplus WG'(S''^0(5)))) \oplus \\ ((\gamma \times S'^0(2) \oplus WG'(S'^0(2)) \oplus (\gamma \times S''^0(2) \oplus WG'(S''^0(2)))).$$

$$(5.3)$$

This shows that the value of $\triangle_2 \oplus \triangle_3$ is determined by $k_{17,...,24}$, $k_{49,...,64}$, $IV'_{9,...,24}$, $IV'_{49,...,56}$, $IV''_{9,...,24}$, $IV''_{49,...,56}$.

From the keystream generation of WG, we know that the first keystream bit is generated from $S^{22}(10)$ (after the key/IV setup, the LFSR is clocked, and $S^{23}(11)$ is used to generate the first keystream bit). If $\triangle_2 \oplus \triangle_3 = 0$, then the first keystream bits for IV' and IV'' should be the same. This property is applied in the attack to determine whether the value of $\triangle_2 \oplus \triangle_3$ is 0.

Assume that the value of $\triangle_2 \oplus \triangle_3$ is randomly distributed, then $\triangle_2 \oplus \triangle_3 = 0$ with probability 2^{-29} . We thus need to generate about 2^{29} pairs $(\triangle_2, \triangle_3)$ in order to obtain a pair satisfying $\triangle_2 \oplus \triangle_3 = 0$. Note that the key is fixed and that $S'^0(2) \oplus S''^0(2) = S'^0(5) \oplus S''^0(5)$ must be satisfied. Three bytes of IV $(IV'_{9,\ldots,24}, IV'_{49,\ldots,56})$ and one-byte difference (\triangle_1) can be freely chosen to generate different $(\triangle_2, \triangle_3)$, so there are about $2^{24} \times 255/2 \approx 2^{31}$ available pairs of $(\triangle_2, \triangle_3)$. Hence there is no problem to generate 2^{29} pairs of $(\triangle_2, \triangle_3)$.

Then we proceed to determine which pair (Δ_2, Δ_3) satisfies $\Delta_2 \oplus \Delta_3 = 0$. For each pair (Δ_2, Δ_3) , we modify the values of $IV'_{1,...,8}$ and $IV''_{1,...,8}$, but we ensure that $IV'_{1,...,8} = IV''_{1,...,8}$. This modification does not affect the value of $\Delta_2 \oplus \Delta_3$, but it affects the value of $S^{22}(10)$. We generate keystream and examine the first keystream bits. If the values of the first keystream bits are the same, then the chance that $\Delta_2 \oplus \Delta_3 = 0$ is improved. In that case, we modify $IV'_{1,...,8}$ and

| | | _ | _ | _ | _ | _ | _ | _ | | | | | | | | | | | | | |
|------------|--------------------|---|--|--|--|--|--|--|---|---|---|---|--|---|---|--|--|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | $^{\triangle_1}$ | 0 | 0 | $^{\bigtriangleup_1}$ | 0 | 0 | 0 | 0 | 0 | 0 | \bigtriangleup_1 | $^{\bigtriangleup_2}$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | 0 | $\substack{ \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 }$ | $\triangle_2 \oplus \triangle_3$ | $\triangle_1 \oplus \triangle_2$ |
| 0 | 0 | 0 | 0 | \bigtriangleup_1 | 0 | 0 | $^{\triangle_1}$ | 0 | 0 | 0 | 0 | 0 | 0 | $^{\bigtriangleup_1}$ | $^{\bigtriangleup}$ 2 | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | 0 | $\substack{\mathbb{A}_1\oplus\mathbb{A}_2\\\oplus\mathbb{A}_3}$ | $\triangle_2 \oplus \triangle_3$ | $\triangle_1 \oplus \triangle_2$ | $\triangle_2 \oplus \triangle_3$ |
| 0 | 0 | 0 | \triangle_1 | 0 | 0 | $^{\Box_1}$ | 0 | 0 | 0 | 0 | 0 | 0 | \bigtriangleup_1 | $^{\bigtriangleup_2}$ | $\triangle_1 \oplus \triangle_2$ | 0 | $\stackrel{\triangle_1 \oplus \triangle_2}{\oplus \triangle_3}$ | $\triangle_2 \oplus \triangle_3$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | $\triangle_2 \oplus \triangle_3$ | \bigtriangledown^3 |
| 0 | 0 | $^{\Box_1}$ | 0 | 0 | \triangle_1 | 0 | 0 | 0 | 0 | 0 | 0 | \bigtriangleup_1 | $^{\bigtriangleup_2}$ | $\bigtriangleup_1\oplus\bigtriangleup_2$ | 0 | $\stackrel{\triangle_1\oplus \triangle_2}{\oplus \triangle_3}$ | $\triangle_2 \oplus \triangle_3$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | $\triangle_2 \oplus \triangle_3$ | $^{\diamond}$ | $\stackrel{\triangle_1 \oplus \triangle_2}{\oplus \triangle_3}$ |
| 0 | \bigtriangleup_1 | 0 | 0 | \bigtriangleup_1 | 0 | 0 | 0 | 0 | 0 | 0 | \bigtriangleup_1 | $^{\bigtriangleup_2}$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | 0 | $\stackrel{\triangle_1 \oplus \triangle_2}{\oplus \triangle_3}$ | $\triangle_2 \oplus \triangle_3$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | $\triangle_2 \oplus \triangle_3$ | $^{\diamond}$ | $\substack{ \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 }$ | $\substack{\triangle_1 \oplus \triangle_2 \\ \oplus \triangle_3}$ |
| $^{\circ}$ | 0 | 0 | $^{\bigtriangleup_1}$ | 0 | 0 | 0 | 0 | 0 | 0 | \bigtriangleup_1 | $^{\bigcirc}2$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | 0 | $\stackrel{\triangle_1\oplus \triangle_2}{\oplus \triangle_3}$ | $\triangle_2 \oplus \triangle_3$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | $\triangle_2 \oplus \triangle_3$ | $^{\triangle_3}$ | $\substack{ \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 }$ | $\substack{ \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 }$ | $\triangle_1 \oplus \triangle_4$ |
| 0 | 0 | $^{\triangle_1}$ | 0 | 0 | 0 | 0 | 0 | 0 | \bigtriangleup_1 | $^{\bigtriangleup}$ | $\triangle_1 \oplus \triangle_2$ | 0 | $\stackrel{\triangle_1\oplus \triangle_2}{\oplus \triangle_3}$ | $\triangle_2 \oplus \triangle_3$ | $\triangle_1 \oplus \triangle_2$ | $\triangle_2 \oplus \triangle_3$ | \bigtriangleup_3 | $\substack{\mathbb{A}_1\oplus\mathbb{A}_2\\\oplus\mathbb{A}_3}$ | $\substack{ \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 }$ | $\triangle_1 \oplus \triangle_4$ | $\begin{array}{c} \bigtriangleup_3 \oplus \bigtriangleup_4 \\ \oplus \bigtriangleup_5 \end{array}$ |
| 0 | $^{\triangle_1}$ | 0 | 0 | 0 | 0 | 0 | 0 | $^{\triangle_1}$ | $^{\bigtriangleup_2}$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | 0 | $\begin{smallmatrix} \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 \end{smallmatrix}$ | $\triangle_2 \oplus \triangle_3$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | $\triangle_2 \oplus \triangle_3$ | \bigtriangleup_3 | $\stackrel{\bigtriangleup_1 \oplus \bigtriangleup_2}{\oplus \bigtriangleup_3}$ | $\substack{\mathbb{A}_1\oplus\mathbb{A}_2\\\oplus\mathbb{A}_3}$ | $\bigtriangleup_1 \oplus \bigtriangleup_4$ | $\substack{\bigcirc 3 \oplus \bigtriangleup_4 \\ \oplus \bigtriangleup_5}$ | $\substack{ \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 \oplus \\ \bigtriangleup_5 \oplus \bigtriangleup_6 }$ |
| $^{\circ}$ | 0 | 0 | 0 | 0 | 0 | 0 | $^{\triangle_1}$ | $^{\diamond}_2$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | 0 | $\stackrel{\triangle_1\oplus\triangle_2}{\oplus \triangle_3}$ | $\triangle_2 \oplus \triangle_3$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | $\triangle_2 \oplus \triangle_3$ | \bigtriangleup_3 | $\stackrel{\triangle_1\oplus\triangle_2}{\oplus\triangle_3}$ | $\substack{ \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 }$ | $\bigtriangleup_1 \oplus \bigtriangleup_4$ | $\substack{\bigcirc 3 \oplus \bigtriangleup 4 \\ \oplus \bigtriangleup 5}$ | $\begin{array}{c} \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 \oplus \\ \bigtriangleup_5 \oplus \bigtriangleup_6 \end{array}$ | $\triangle_4 \oplus \triangle_6$ |
| 0 | 0 | 0 | 0 | 0 | 0 | $^{\triangle_1}$ | $^{\triangle_2}$ | $\triangle_1 \oplus \triangle_2$ | 0 | $\stackrel{\triangle_1\oplus\triangle_2}{\oplus \triangle_3}$ | $\triangle_2 \oplus \triangle_3$ | $\bigtriangleup_1 \oplus \bigtriangleup_2$ | $\triangle_2 \oplus \triangle_3$ | $^{\triangle_3}$ | $\stackrel{	riangle 1}{\oplus} \stackrel{	riangle 2}{\oplus} \stackrel{	riangle 2}{\oplus}$ | $\stackrel{\triangle_1\oplus\triangle_2}{\oplus \triangle_3}$ | $\bigtriangleup_1 \oplus \bigtriangleup_4$ | $\substack{\bigcirc 3 \oplus \bigtriangleup 4 \\ \oplus \bigtriangleup 5}$ | $\begin{array}{c} \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 \oplus \\ \bigtriangleup_5 \oplus \bigtriangleup_6 \end{array}$ | $\triangle_4 \oplus \triangle_6$ | $\stackrel{\bigtriangleup_4 \oplus \bigtriangleup_5}{\oplus \bigtriangleup_7}$ |
| 0 | 0 | 0 | 0 | 0 | \triangle_1 | $^{\triangle_2}$ | $\triangle_1 \oplus \triangle_2$ | 0 | $\stackrel{	riangle _1 \oplus 	riangle _2 }{\oplus 	riangle _3 }$ | $\triangle_2 \oplus \triangle_3$ | $\triangle_1 \oplus \triangle_2$ | $\triangle_2 \oplus \triangle_3$ | \bigtriangleup_3 | $\stackrel{\triangle_1 \oplus \triangle_2}{\oplus \triangle_3}$ | $\bigcirc 1 \oplus \bigcirc 2 \oplus \bigcirc 3 \oplus \bigcirc 3$ | $\bigtriangleup_1 \oplus \bigtriangleup_4$ | $\substack{\bigtriangleup_3 \oplus \bigtriangleup_4 \\ \oplus \bigtriangleup_5}$ | $\begin{array}{c} \bigtriangleup_1 \oplus \bigtriangleup_2 \\ \oplus \bigtriangleup_3 \oplus \\ \bigtriangleup_5 \oplus \bigtriangleup_6 \end{array}$ | $	riangle_4 \oplus 	riangle_6$ | $\substack{\triangle_4 \oplus \triangle_5\\\oplus \triangle_7}$ | $\begin{array}{c} \mathbb{C}_2 \oplus \mathbb{C}_3 \\ \oplus \mathbb{C}_4 \oplus \\ \mathbb{C}_5 \oplus \mathbb{C}_6 \\ \oplus \mathbb{C}_7 \\ \oplus \mathbb{C}_8 \end{array}$ |
| tep 1 | tep 2 | tep 3 | tep 4 | tep 5 | tep 6 | step 7 | tep 8 | tep 9 | tep 10 | tep 11 | tep 12 | tep 13 | tep 14 | tep 15 | tep 16 | tep 17 | tep 18 | tep 19 | tep 20 | tep 21 | cep 22 |
| | | $\begin{array}{c ccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{c ccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{c ccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{c ccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{cccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{cccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{cccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{cccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{cccccccccccccccccccccccccccccccccccc$ | | | $ \begin{array}{cccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{cccccccccccccccccccccccccccccccccccc$ | $ \begin{array}{cccccccccccccccccccccccccccccccccccc$ |

Table 5.1: The differential propagation in the key/IV setup of WG $\,$

 $IV_{1,\ldots,8}''$ again and observe the first keystream bits. This process ends when the first keystream bits are not the same or this process is repeated for 40 times. If one $(\triangle_2, \triangle_3)$ passes the test for 40 times, then we know that $\triangle_2 \oplus \triangle_3 = 0$ with probability extremely close to 1. (Each wrong pair could pass this filtering process with probability 2^{-40} . One pair of 2^{29} wrong pairs could pass this process with probability 2^{-11} .) Thus with about $2 \times 2^{29} \times \sum_{i=1}^{40} \frac{i}{2^i} = 2^{31}$ chosen IVs, we can find a pair $(\triangle_2, \triangle_3)$ satisfying $\triangle_2 \oplus \triangle_3 = 0$. Subsequently according to Eqn. (3) and $\triangle_2 \oplus \triangle_3 = 0$, we recover 24 bits of the secret key, $k_{17,\ldots,24}$ and $k_{49,\ldots,64}$.

The above attack can be improved if we consider the differences at $S^{22}(7)$ and $S^{22}(8)$. The differences there are both $\triangle_1 \oplus \triangle_2 \oplus \triangle_3$. If the value of $\triangle_1 \oplus \triangle_2 \oplus \triangle_3$ is 0, then the third and fourth bits of the two keystreams would be the same. If we only observe the third and fourth keystream bits, then $k_{17,...,24}$ and $k_{49,...,64}$ can be recovered with $2 \times 2^{29} \times \sum_{i=1}^{20} \left(\frac{1}{2^{i-1}} - \frac{1}{2^i}\right) \times i = 2^{30.4}$ chosen IVs. In the attack, we observe the first, third and fourth keystream bits, then

In the attack, we observe the first, third and fourth keystream bits, then recovering $k_{17,\ldots,24}$ and $k_{49,\ldots,64}$ requires about $2 \times 2^{28} \times 2^{1.13} = 2^{30.1}$ chosen IVs (the value $2^{1.13}$ is obtained through numerical computation).

By setting the difference at $S^{0}(3)$ and $S^{0}(6)$ and observing the second and third bits of the keystream, we can recover another 24 bits of the secret key, $k_{25,...,40}$ and $k_{65,...,72}$. We need $2^{30.4}$ chosen IVs.

So with about $2^{30.1} + 2^{30.4} = 2^{31.3}$ chosen IVs, we can recover 48 bits of the 80-bit secret key. It shows that the key/IV setup of the WG stream cipher is insecure.

5.3.2 Attacks on WG with key and IV sizes larger than 80 bits

The WG ciphers with the key and IV sizes larger than 80 bits are all vulnerable to the chosen IV attacks. The attacks are very similar to the above attack. We omit the details of the attacks here. The results are given below:

- 1. For WG with 96-bit key and 96-bit IV, 48 bits of the key can be recovered with complexity about the same as the above attack.
- 2. For WG with IV sizes larger than 96 bits, 72 bits of the key can be recovered with complexity about 1.5 times that of the above attack.

5.3.3 Attacks on WG with 64-bit IV size

We use WG with an 80-bit key and a 64-bit IV as an example to illustrate the attack. For WG cipher with an 80-bit key and a 64-bit IV, the key and IV are loaded into the LFSR as follows:

| $S_{1,,16}(1) = k_{1,,16}$ | $S_{1,,16}(2) = k_{17,,32}$ |
|--|--|
| $S_{1,,16}(3) = k_{33,,48}$ | $S_{1,,16}(4) = k_{49,,64}$ |
| $S_{1,,16}(5) = k_{65,,80}$ | $S_{1,,16}(9) = k_{1,,16}$ |
| $S_{1,,16}(10) = k_{17,,32} \oplus 1$ | $S_{1,,16}(11) = k_{33,,48}$ |
| $\begin{split} S_{17,,24}(1) &= IV_{1,,8} \\ S_{17,,24}(3) &= IV_{17,,24} \\ S_{17,,24}(5) &= IV_{33,,40} \\ S_{17,,24}(7) &= IV_{49,,56} \end{split}$ | $\begin{split} S_{17,\dots,24}(2) &= IV_{9,\dots,16}\\ S_{17,\dots,24}(4) &= IV_{25,\dots,32}\\ S_{17,\dots,24}(6) &= IV_{41,\dots,48}\\ S_{17,\dots,24}(8) &= IV_{57,\dots,64} \end{split}$ |

In the attack, we introduce differences at S(2) and S(5), but we can only generate about 2^{23} pairs of $(\triangle_2, \triangle_3)$ since we can only modify $IV_{9,...,16}$ and $IV_{33,...,40}$. Thus we can obtain a pair $(\triangle_2, \triangle_3)$ satisfying $\triangle_2 \oplus \triangle_3 = 0$ or $\triangle_1 \oplus \triangle_2 \oplus \triangle_3 = 0$ with probability 2^{-5} . Once we know $\triangle_2 \oplus \triangle_3 = 0$ or $\triangle_1 \oplus \triangle_2 \oplus \triangle_3 = 0$, we can recover 29 bits of information on $k_{17,...,32}$ and $k_{65,...,80}$. It shows that 29 bits of information of the secret key can be recovered with probability 2^{-5} . This attack requires about $2^{25.1}$ chosen IVs.

The attack on WG with 96-bit key and 64-bit IV is similar to the above attack. We introduce differences at S(2) and S(5) or at S(3) and S(6). In the attack 29 bits of information on $k_{17,...,32}$ and $k_{65,...,80}$ can be recovered with probability 2^{-5} , and another 29 bits of information on $k_{33,...,48}$ and $k_{81,...,96}$ can be recovered with probability 2^{-5} .

The attack on WG with 112-bit key and 64-bit IV is also similar. The result is that 29 bits of information on $k_{17,...,32}$ and $k_{65,...,80}$ can be recovered with probability 2^{-5} , 29 bits of information on $k_{33,...,48}$ and $k_{81,...,96}$ can be recovered with probability 2^{-5} , and 29 bits of information on $k_{49,...,64}$ and $k_{97,...,112}$ can be recovered with probability 2^{-5} .

The attack on WG with 128-bit key and 64-bit IV is also similar. The result is that 29 bits of information on $k_{17,...,32}$ and $k_{65,...,80}$ can be recovered with probability 2^{-5} , 29 bits of information on $k_{33,...,48}$ and $k_{81,...,96}$ can be recovered with probability 2^{-5} , 29 bits of information on $k_{49,...,64}$ and $k_{97,...,112}$ can be recovered with probability 2^{-5} , and 29 bits of information on $k_{64,...,80}$ and $k_{113,...,128}$ can be recovered with probability 2^{-5} .

5.4 Conclusion

In this chapter, we show that the resynchronization mechanisms of WG is vulnerable to a differential attack. It shows that a differential attack is powerful in analyzing the non-linear resynchronization mechanism of a stream cipher.

The designers of WG recommended to use 44 steps in the initialization to resist a differential attack [104]. It is a small modification to the design to achieve

a secure key/IV setup. However, it is inefficient. We recommend to change the primitive polynomial tap positions so that the tap distances are coprime, and to generate the first keystream bit from S(1) instead of S(10). Then we expect that WG with 22-step key/IV setup will be able to resist a differential attack.

Chapter 6

Resynchronization Attack III Slide Attack on LEX

Abstract. LEX is a stream cipher submitted to eStream – the ECRYPT stream cipher project. In this chapter, we point out a security flaw in its resynchronization. The resynchronization of LEX is vulnerable to a slide attack. If a key is used with about $2^{60.8}$ random IVs, and 20,000 keystream bytes are generated from each IV, then the key of the strong version of LEX could be recovered easily with a slide attack. The resynchronization attack on LEX shows that block cipher related attacks are powerful in analyzing non-linear resynchronization mechanisms.

6.1 Introduction

LEX [20] is a stream cipher submitted by Biryukov to eStream, the ECRYPT stream cipher project [45]. The keystream generation of LEX is based on the Advanced Encryption Standard [101]. However, the resynchronization mechanism of LEX is insecure and is vulnerable to slide attack [23]. Breaking the strong version of LEX requires about $2^{60.8}$ random IVs.

This chapter is organized as follows. LEX is introduced in Sect. 6.2. The slide attack on LEX is described in Sect. 6.3. Section 6.4 concludes this chapter.

6.2 Description of LEX

LEX is based on the block cipher AES. The keystream bits are generated by extracting 32 bits from each round of AES in the 128-bit Output Feedback (OFB) mode [98]. LEX is about 2.5 times faster than AES. Fig. 6.1 [20] shows how the

65

AES is initialized and chained. First a standard AES key-schedule for a secret 128-bit key K is performed. Then a given 128-bit IV is encrypted by a single AES invocation: $S = AES_K(IV)$. The S and the subkeys are the output of the initialization process.

S is encrypted by K in the 128-bit OFB mode (for the more secure variant, K is changed every 500 AES encryptions). At each round, 32 bits of the middle value of AES are extracted to form the keystream. The bytes $b_{0,0}$, $b_{0,2}$, $b_{2,0}$, $b_{2,2}$ at every odd round and the bytes $b_{0,1}$, $b_{0,3}$, $b_{2,1}$, $b_{2,3}$ at every even round are selected, as shown in Fig. 6.2 [20].



Figure 6.1: Initialization and stream generation



Figure 6.2: The positions of the output extracted in the even and odd rounds

6.3 Slide Attack on the Resynchronization of LEX

The security of LEX depends heavily on the fact that only a small amount of information is released for each round (including the input and output) of AES.

The slide attack intends to retrieve all the information of one AES round input (or output) in LEX.

Denote $S_i = E_K^i(IV)$, where $E^i(m)$ means that m is encrypted i times, $S_0 = IV$; denote the 320-bit keystream extracted from the *i*-th encryption as k_i for $i \ge 2$. For two IVs, IV' and IV'', if $k'_2 = k''_j$ (j > 2), then we know that $S'_1 = S''_{j-1}$. Immediately, we know that $S''_{j-2} = S'_0 = IV'$. Note that k''_{j-1} is extracted from $E_K(S''_{j-2})$, so k''_{j-1} is extracted from $E_K(IV')$; this means that we know the input to AES, and we know 32 bits from the output of the first round. In the following, we show that it is easy to recover the secret key from this 32 bits of information of the first round output.

Denote the 16-byte output of the r-th round of AES with $m_{i,j}^r$ $(0 \le i, j \le 3)$, and denote the 16-byte round key at the end of the r-th round with $w_{i,j}^r$ $(0 \le i, j \le 3)$. Now if $m_{0,0}^1$, $m_{0,2}^1$, $m_{2,0}^1$, $m_{2,2}^1$ are known, i.e., four bytes of the first round output are known, then we obtain the following four equations:

$$m_{0,0}^{1} \oplus w_{0,0}^{1} = \operatorname{MixColumn}((m_{0,0}^{0} \oplus w_{0,0}^{0}) \parallel (m_{1,3}^{0} \oplus w_{1,3}^{0}) \\ \parallel (m_{2,2}^{0} \oplus w_{2,2}^{0}) \parallel (m_{3,1}^{0} \oplus w_{3,1}^{0})) \& 0xFF$$

$$(6.1)$$

$$m_{2,0}^{*} \oplus w_{2,0}^{*} = (\text{MixColumn}((m_{0,0}^{*} \oplus w_{0,0}^{*}) \parallel (m_{1,3}^{*} \oplus w_{1,3}^{*}))$$

$$\| (m_{2,2}^{0} \oplus w_{2,2}^{0}) \| (m_{3,1}^{0} \oplus w_{3,1}^{0}) >> 16) \& 0xFF \quad (6.2)$$

$$_{,2} \oplus w_{0,2}^{1} = \text{MixColumn}((m_{0,2}^{0} \oplus w_{0,2}^{0}) \| (m_{1,1}^{0} \oplus w_{1,1}^{0})$$

$$\| (m_{2,0}^0 \oplus w_{2,0}^0) \| (m_{3,3}^0 \oplus w_{3,3}^0)) \& 0xFF$$
(6.3)

$$\begin{split} m_{2,2}^{1} \oplus w_{2,2}^{1} &= (\text{MixColumn}((m_{0,2}^{0} \oplus w_{0,2}^{0}) \parallel (m_{1,1}^{0} \oplus w_{1,1}^{0}) \\ &\parallel (m_{2,0}^{0} \oplus w_{2,0}^{0}) \parallel (m_{3,3}^{0} \oplus w_{3,3}^{0}) >> 16) \& 0xFF. \end{split}$$

Each equation leaks one byte of information on the secret key. In the above four equations, 12 bytes of the subkey are involved. To recover all these 12 bytes, we need three inputs to AES and the related 32-bit first round outputs so that we can obtain 12 equations. These 12 equations can be solved with about $\alpha \times 2^{32}$ operations, where α is a small constant. With 96 bits of the key have been recovered, the rest of the 32 bits of AES can be recovered by exhaustive search.

We now compute the number of IVs required to generate three collisions. Suppose that a secret key is used with about $2^{65.3}$ random IVs, and each IV^i is used to generate a 640-bit keystream k_2^i, k_3^i . Since the block size of AES is 128 bits, we know that with high probability there are three collisions $k_2^i = k_3^j$ for different *i* and *j* since $\frac{2^{65.3} \times (2^{65.3}-1)}{2} \times 2^{-128} \approx 3$.

The number of IVs could be reduced if more keystream bits are generated from each IV. In [20], it is suggested to change the key every 500 AES encryptions for a strong variant of LEX. Suppose that each IV is applied to generate 500 320-bit outputs, then with $2^{60.8}$ IVs, we could find three collisions $k_2^i = k_x^i$ (2 < x < 500) and recover the key of LEX. For the original version of LEX, the AES key is not changed during the keystream generation. Suppose that each IV is used to generate 2^{50} keystream bytes, then the key could be recovered with about 2^{43} random IVs (here we need to consider that the state update function of LEX is reversible; otherwise, the amount of IV required in the attack could be greatly reduced).

For a secure stream cipher with a 128-bit key and a 128-bit IV, each key would never be recovered faster than exhaustive key search no matter how many IVs are used together with that key. But for LEX each key could be recovered faster than exhaustive search if that key is used together with about 2^{61} random IVs. We thus conclude that LEX is theoretically insecure.

For a stream cipher with 128-bit key and 128-bit IV, if the attacker can choose the IV, then one of 2^{64} keys could be recovered with about 2^{64} pre-computations (based on the birthday paradox). The complexity of such an attack is close to our attack on LEX. However, there are two major differences between these two attacks. One difference is that the attack based on birthday paradox is a chosen IV attack while our attack is a random IV attack. Another difference is that the attack based on the birthday paradox results in the recovery of one of nkeys, while our attack recovers one particular key. Recovering one of n keys and recovering one particular key are two different types of attacks being used in different scenarios, so it is not meaningful to simply compare their complexities.

6.4 Conclusion

In this chapter, we show that the resynchronization mechanisms of LEX is vulnerable to a slide attack that is develoed against block cipher. It is not so surprising since LEX is based on a block cipher.

Chapter 7

Resynchronization Attack IV Differential Attack on Py, Py6 and Pypy

Abstract. Py and Pypy are efficient array-based stream ciphers designed by Biham and Seberry. Both were submitted to the eSTREAM competition. This chapter shows that Py and Pypy are practically insecure. If one key is used with about 2^{16} IVs with special differences, with high probability two identical keystreams will appear. This can be exploited in a key recovery attack. For example, for a 16-byte key and a 16-byte IV, 2^{23} chosen IVs can reduce the effective key size to 3 bytes. For a 32-byte key and a 32-byte IV, the effective key size is reduced to 3 bytes with 2^{24} chosen IVs. Py6, a variant of Py, is more vulnerable to these attacks.

7.1 Introduction

RC4 has inspired the design of a number of fast stream ciphers, such as ISAAC [72], Py [25], Pypy [26] and MV3 [78]. RC4 was designed by Rivest in 1987. Being the most widely used software stream cipher, RC4 is extremely simple and efficient. At the time of the invention of RC4, its array based design was completely different from the previous stream ciphers mainly based on linear feedback shift registers.

There are two main motives to improve RC4. One motive is that RC4 is byte oriented, so we need to design stream ciphers that can run more efficiently on today's 32-bit microprocessors. Another motive is to strengthen RC4 against various attacks [59, 82, 95, 51, 50, 86, 94, 106, 84, 85]. Two of these attacks affect the security of RC4 in practice: the broadcast attack which exploits the weakness that the first few keystream bytes are heavily biased [86], and the key

69

recovery attack using related IVs [50] which results in the practical attack on RC4 in WEP [84]. These two serious weaknesses are caused by the imperfection in the initialization of RC4.

Recently Biham and Seberry proposed the stream cipher Py [25] which is related to the design of RC4. Py is one of the fastest stream ciphers on 32-bit processors (about 2.5 times faster than RC4). A distinguishing attack against Py was found by Paul, Preneel and Sekar [109]. In that attack, the keystream can be distinguished from random with about 2^{88} bytes. Later, the attack was improved by Crowley [37], and the data required in the attack is reduced to 2^{72} . In order to resist the distinguishing attack on Py, the designers of Py decided to discard half of the outputs, i.e., the first output of the two outputs at each step is discarded. The new version is called Pypy [26]. Py and Pypy are selected as focus ciphers in the Phase 2 of the ECRYPT eSTREAM project.

The initializations of Py and Pypy are identical. In this chapter, we show that there are serious flaws in the initialization of Py and Pypy, thus these two ciphers are vulnerable to differential cryptanalysis [27]. Two keystreams can be identical if a key is used with about 2^{16} IVs with special differences. It is a practical threat since the set of IVs required in the attack may appear with high probability in applications. Then we show that part of the key of Py and Pypy can be recovered with chosen IVs. For a 16-byte key and a 16-byte IV, 2^{23} chosen IVs can reduce the effective key size to 3 bytes.

Py6 [25] is a variant of Py with reduced internal state size. We show that Py6 is more vulnerable to the attacks against Py and Pypy.

This chapter is organized as follows. In Sect. 7.2, we illustrate the key and IV setups of Py and Pypy. Section 7.3 describes the attack of generating identical keystreams. The key recovery attack is given in Sect. 7.4. In Sect. 7.5, we outline the attacks against Py6. Section 7.6 concludes this chapter.

7.2 The Specifications of Py and Pypy

Py and Pypy are two synchronous stream ciphers supporting key and IV sizes up to 256 bytes and 64 bytes, respectively. The initializations of Py and Pypy are identical. The initialization consists of two stages: key setup and IV setup.

In the following descriptions, P is an array with 256 8-bit elements. Y is an array with 260 32-bit elements, s is a 32-bit integer. *YMININD* = -3, *YMAXIND* = 256. The table 'internal_permutation' is a constant permutation table with 256 elements. ' \wedge ' and '&' in the pseudo codes denote binary XOR and AND operations, respectively. 'u8' and 'u32' mean 'unsigned 8-bit integer' and 'unsigned 32-bit integer', respectively. 'ROTL32(a,n)' means that the 32-bit a is left rotated over n bits.

7.2.1 The key setup

The key setups of Py and Pypy are identical. In the key setup, the key is used to initialize the array Y. The description is given below.

```
keysizeb=size of key in bytes;
ivsizeb=size of IV in bytes;
YMININD = -3; YMAXIND = 256;
s = internal_permutation[keysizeb-1];
s = (s<<8) | internal_permutation[(s ^ (ivsizeb-1))&OxFF];</pre>
s = (s<<8) | internal_permutation[(s ^ key[0])&0xFF];</pre>
s = (s<<8) | internal_permutation[(s ^ key[keysizeb-1])&0xFF];</pre>
for(j=0; j<keysizeb; j++)</pre>
{
     s = s + key[j];
     s0 = internal_permutation[s&0xFF];
     s = ROTL32(s, 8) ^ (u32)s0;
}
/* Again */
for(j=0; j<keysizeb; j++)</pre>
{
     s = s + key[j];
     s0 = internal_permutation[s&OxFF];
     s ^= ROTL32(s, 8) + (u32)s0;
}
/* Algorithm C is the following 'for' loop */
for(i=YMININD, j=0; i<=YMAXIND; i++)</pre>
{
     s = s + key[j];
     s0 = internal_permutation[s&OxFF];
     Y(i) = s = ROTL32(s, 8) ^ (u32)s0;
     j = (j+1) mod keysizeb;
}
```

7.2.2 The IV setup

The IV setups of Py and Pypy are identical. In the IV setup, the IV is used to affect every bit of the internal state. EIV is a temporary byte array with the same size as the IV. The IV setup is given below.

/* Create an initial permutation */
u8 v= iv[0] ^ ((Y(0)>>16)&0xFF);

```
u8 d=(iv[1 mod ivsizeb] ^ ((Y(1)>>16)&OxFF))|1;
for(i=0; i<256; i++)</pre>
{
     P(i)=internal_permutation[v];
     v+=d;
}
/* Now P is a permutation */
/* Initial s */
s = ((u32)v << 24)^{((u32)d << 16)^{((u32)P(254)}<< 8)^{((u32)P(255))}};
s ^= Y(YMININD)+Y(YMAXIND);
/* Algorithm A is the following 'for' loop */
for(i=0; i<ivsizeb; i++)</pre>
{
     s = s + iv[i] + Y(YMININD+i);
     u8 s0 = P(s\&OxFF);
     EIV(i) = s0;
     s = ROTL32(s, 8) ^ (u32)s0;
}
/* Again, but with the last words of Y, and update EIV */
/* Algorithm B is the following 'for' loop */
for(i=0; i<ivsizeb; i++)</pre>
{
     s = s + iv[i] + Y(YMAXIND-i);
     u8 s0 = P(s\&OxFF);
     EIV(i) += s0;
     s = ROTL32(s, 8) ^ (u32)s0;
}
/*updating the rolling array and s*/
for(i=0; i<260; i++)</pre>
{
     u32 x0 = EIV(0) = EIV(0)^{(s&0xFF)};
     rotate(EIV);
     swap(P(0),P(x0));
     rotate(P);
     Y(YMININD)=s=(s^Y(YMININD))+Y(x0);
     rotate(Y);
}
s=s+Y(26)+Y(153)+Y(208);
if(s==0)
```

s=(keysizeb*8)+((ivsizeb*8)<<16)+0x87654321;</pre>

7.2.3 The keystream generation

After the key and IV setup, the keystream is generated. One step of the keystream generation of Py is given below. Note that the first output at each step is discarded in Pypy.

```
/* swap and rotate P */
swap(P(0), P(Y(185)&0xFF));
rotate(P);
/* Update s */
s+=Y(P(72)) - Y(P(239));
s=ROTL32(s, ((P(116) + 18)&31));
/* Output 8 bytes (least significant byte first) */
output ((ROTL32(s, 25) ^ Y(256)) + Y(P(26)));
output (( s ^ Y(-1)) + Y(P(208)));
/* Update and rotate Y */
Y(-3)=(ROTL32(s, 14) ^ Y(-3)) + Y(P(153));
rotate(Y);
```

7.3 Identical Keystreams

We notice that the IV appears only in the IV setup algorithm described in Sect. 7.2.2. At the beginning of the IV setup, only 15 bits of the IV (iv[0] and iv[1]) are applied to initialize the array P and s (the least significant bit of iv[1] is not used). For an IV pair, if those 15 bits are identical, then the resulting P are the same. Then we notice that the IV is applied to update s and EIV as follows.

```
for(i=0; i<ivsizeb; i++)
{
    s = s + iv[i] + Y(YMININD+i);
    u8 s0 = P(s&0xFF);
    EIV(i) = s0;
    s = ROTL32(s, 8) ^ (u32)s0;
}
for(i=0; i<ivsizeb; i++)</pre>
```

```
{
    s = s + iv[i] + Y(YMAXIND-i);
    u8 s0 = P(s&0xFF);
    EIV(i) += s0;
    s = ROTL32(s, 8) ^ (u32)s0;
}
```

We call the first 'for' loop Algorithm A, and the second 'for' loop Algorithm B. In the following, we give two types of IV pairs that result in identical keystreams.

7.3.1 IVs differing in two bytes

We illustrate the attack with an example. Suppose that two IVs, iv_1 and iv_2 , differing in only two consecutive bytes with $iv_1[i] \oplus iv_2[i] = 1$, the least significant bit of $iv_1[i]$ is 1, $iv_1[i+1] \neq iv_2[i+1]$ ($1 \leq i \leq ivsizeb - 1$), and $iv_1[j] = iv_2[j]$ for $0 \leq j < i$ and $i+1 < j \leq ivsizeb - 1$. We trace how the difference in IV affects s and EIV in Algorithm A. At the *i*th step in Algorithm A,

```
s = s + iv[i] + Y(YMININD+i);
u8 s0 = P(s&0xFF);
EIV(i) = s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

At the end of the *i*th step, $EIV_1[i] \neq EIV_2[i]$. Let $\beta_1 = EIV_1[i]$, and $\beta_2 = EIV_2[i]$. We obtain that $s_1 - s_2 = 256 + \delta_1$, where $\delta_1 = (\beta_1 \oplus x) - (\beta_2 \oplus x)$, and x = ROTL32(s, 8). Then we look at the next step.

```
s = s + iv[i+1] + Y(YMININD+i+1);
u8 s0 = P(s&0xFF);
EIV(i+1) = s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

Because $iv_1[i+1] \neq iv_2[i+1]$, if $iv_2[i+1] - iv_1[i+1] = \delta_1$, then s_1 and s_2 become identical with high probability. Let $s_1 = s_2$ with probability p_1 . Based on the simulation, we obtain that $p_1 = 2^{-10.6}$. If $s_1 = s_2$, then $EIV_1[i+1] = EIV_2[i+1]$, and in the following steps $i+2, i+3, \cdots, i+ivsizeb-1$ in Algorithm A, s_1 and s_2 remain the same, and $EIV_1[j] = EIV_2[j]$ for $j \neq i$.

After Algorithm A, the iv[i] and iv[i+1] are used again to update s and EIV in Algorithm B. At the *i*th step in Algorithm B,

```
s = s + iv[i] + Y(YMAXIND-i);
u8 s0 = P(s&0xFF);
EIV(i) += s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

At the end of this step, $EIV_1[i] = EIV_2[i]$ with probability $\frac{1}{255}$. Let $\gamma_1 = s0_1$, and $\gamma_2 = s0_2$. If $EIV_1[i] = EIV_2[i]$, we know that $\gamma_2 - \gamma_1 = \beta_1 - \beta_2$. At the end of this step, $s_1 - s_2 = 256 + \delta_2$, where $\delta_2 = (\gamma_1 \oplus y) - (\gamma_2 \oplus y)$, and y is ROTL32(s,8). Note that δ_1 and δ_2 are correlated since $\gamma_2 - \gamma_1 = \beta_1 - \beta_2$. Then we look at the next step.

```
s = s + iv[i+1] + Y(YMAXIND-i-1);
u8 s0 = P(s&0xFF);
EIV(i+1) += s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

At the end of this step, if $iv_2[i+1] - iv_1[i+1] = \delta_2$, then s_1 and s_2 become identical with high probability. Note that $iv_2[i+1] - iv_1[i+1] = \delta_1$, and δ_1 and δ_2 are correlated, so $iv_2[i+1] - iv_1[i+1] = \delta_2$ with probability larger than 2^{-8} . Let $s_1 = s_2$ with probability p'_1 . Based on a simulation, we obtain that $p'_1 = 2^{-5.6}$. Once the two *s* values are identical, $EIV_1[i+1] = EIV_2[i+1]$, and in the following steps $i+2, i+3, \cdots, i+ivsize-1$ in Algorithm B, s_1 and s_2 remain the same, and $EIV_1[i+2] = EIV_2[i+2]$, $EIV_1[i+3] = EIV_2[i+3], \cdots$, $EIV_1[i+ivsize-1] = EIV_2[i+ivsize-1]$.

Thus after introducing the IV to update s and EIV, $s_1 = s_2$ and $EIV_1 = EIV_2$ with probability $p_1 \times \frac{1}{255} \times p'_1 \approx 2^{-24.2}$.

Note that once an IV has been introduced in Algorithm A and B, the IV is not used in the rest of the IV setup. Thus once $s_1 = s_2$ and $EIV_1 = EIV_2$ at the end of Algorithm B, we know that those two keystreams will be the same.

Experiment 7.1. We use 2^{14} random 128-bit keys in the attack. For each key, we randomly generate 2^{16} pairs of 128-bit IV that differ in only two bytes: $iv_1[6] \oplus iv_2[6] = 1$, $iv_1[7] \neq iv_2[7]$. We found that 111 pairs of those 2^{30} keystream pairs are identical. For example, for the key (08 da f2 35 a3 d5 94 e2 85 cc 68 ba 7e 10 8a b4), and the IV pair (6e e7 09 b1 35 85 2f 07 1a fe 3f 50 a8 84 30 11) and (6e e7 09 b1 35 85 2e 80 1a fe 3f 50 a8 84 30 11), the two keystreams are identical, and the first 16 keystream bytes of Pypy are (6f eb ca 18 54 3f 59 96 b6 17 8a 54 6e bd 45 1f).

From the experiment, we deduce that for an IV pair with the required difference, the two keystreams are identical with probability about $\frac{111}{2^{30}} = 2^{-23.2}$, about twice the theoretical value.

The IV difference at two bytes. In the above analysis, the difference is chosen as $iv_1[i] \oplus iv_2[i] = 1$, $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 1)$. We can generalize this type of IV difference so that $iv_1[i]$ and $iv_2[i]$ can take other differences. As long as $(iv_1[i] - iv_2[i]) \mod 256 = 1$ or 255, $iv_1[i+1] \neq iv_2[i+1]$ $(i \ge 2)$, there is a non-zero probability that the two keystreams can be identical.

For example, if $iv_1[i] \oplus iv_2[i] = 3$, the two least significant bits of $iv_1[i]$ are 01 or 10, and $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 2)$, then two identical keystreams appear with probability $2^{-23.2}$. On average, if $iv_1[i] - iv_2[i] = 1$, and $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 2)$, then two identical keystreams appear with probability $2^{-26.4}$.

7.3.2 IVs differing in three bytes

In the above attack, we deal with the *i*th and (i + 1)th bytes of the IV, and use the difference at iv[i + 1] to eliminate the difference introduced by iv[i] in *s*. In the following, we introduce another type of difference to deal with the situation when the difference at iv[i+1] cannot eliminate the difference introduced by iv[i]in *s*. The solution is to introduce a difference in iv[i+4].

We illustrate the attack with an example. Suppose that two IVs, iv_1 and iv_2 , differ in only three bytes $iv_1[i] \oplus iv_2[i] = 0x80$, the most significant bit of $iv_1[i]$ is 1, $iv_1[i+1] \neq iv_2[i+1]$, $iv_1[i+4] \oplus iv_2[i+4] = 0x80$, and the most significant bit of $iv_1[i+4]$ is 0, where $i \geq 2$. We trace how the difference affects s and EIV. At the *i*th step in Algorithm A,

```
s = s + iv[i] + Y(YMININD+i);
u8 s0 = P(s&0xFF);
EIV(i) = s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

At the end of this step, $EIV_1[i] \neq EIV_2[i]$, and $s_1 - s_2 = 0x8000 + \delta_1$, where δ_1 is the difference of two different 8-bit numbers. Then we look at the next step.

s = s + iv[i+1] + Y(YMININD+i+1); u8 s0 = P(s&0xFF); EIV(i+1) = s0; s = ROTL32(s, 8) ^ (u32)s0;

Because $iv_1[i+1] \neq iv_2[i+1]$, $s_1 - s_2 = 0x8000$ with probability $p_2 = 2^{-8}$. If $s_1 - s_2 = 0x8000$, then $EIV_1[i+1] \oplus EIV_2[i+1] = 0$.

Since $v_1[i+2] = v_2[i+2]$, at the end of the (i+2)th step of Algorithm A, $EIV_1[i+2] = EIV_2[i+2]$, and $s_1 - s_2 = 0x800000$ with probability close to 1.

Since $v_1[i+3] = v_2[i+3]$, at the end of the (i+3)th step of Algorithm A, $EIV_1[i+3] = EIV_2[i+3]$, and $s_1 - s_2 = 0x80000000$ with probability close to 1. Now consider the (i+4)th step.

```
s = s + iv[i+4] + Y(YMININD+i+4);
u8 s0 = P(s&OxFF);
```

EIV(i+4) = s0; s = ROTL32(s, 8) ^ (u32)s0;

At the end of this step, the probability that $EIV_1[i+4] = EIV_2[i+4]$, and $s_1 = s_2$ is 1. So for the above 5 steps, $s_1 = s_2$ with probability p_2 . Once $s_1 = s_2$, in the following steps $i + 5, i + 6, \dots, i + ivsize - 1$ in Algorithm A, the s_1 and s_2 remain the same, and $EIV_1[i+5] = EIV_2[i+5]$, $EIV_1[i+6] = EIV_2[i+6]$, $\dots, EIV_1[i+ivsize-1] = EIV_2[i+ivsize-1]$.

Then iv[i] and iv[i+1] are used again to update s and EIV. With a similar analysis, we can show that at the end of the updating, $EIV_1 = EIV_2$, $s_1 = s_2$ with probability about $(p_2)^2 \times \frac{1}{255} \approx 2^{-24}$. (As shown in the experiment in the next subsection, this probability is about $2^{-22.9}$.)

The IV difference at three bytes. In the above analysis, the difference is chosen at only three bytes, $iv_1[i] \oplus iv_2[i] = 0x80$, the most significant bit of $iv_1[i]$ is 1, $iv_1[i+1] \neq iv_2[i+1]$, $iv_1[i+4] \oplus iv_2[i+4] = 0x80$, and the most significant bit of $iv_1[i+4]$ is 0 $(i \geq 2)$. For this type of IV difference, we can generalize it so that $iv_1[i]$ and $iv_2[i]$ can choose other differences instead of 0x80. In fact, once we set the difference as $iv_1[i] - iv_2[i] = iv_2[i+4] - iv_1[i+4]$, $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 2)$, then the two keystreams are identical with probability close to 2^{-23} . For two IVs different only at three bytes, if $iv_1[1] \oplus iv_2[1] = 1$, $iv_1[2] \neq iv_2[2]$, and $iv_1[1] - iv_2[1] = iv_2[5] - iv_1[5]$, then this IV pair is also weak.

7.3.3 Improving the attack

The number of IVs required to generate identical keystreams can be reduced in practice. The idea is to generate more IV pairs from a group of IVs. For the IV pair with a two-byte difference $iv_1[i] \oplus iv_2[i] = 1$, $iv_1[i+1] \neq iv_2[i+1]$, if iv[2] takes all the 256 values, then we can obtain $255 \times 255 = 2^{15.99}$ IV pairs with the required differences from 512 IVs. Thus with 512 chosen IVs, the probability that there is one pair of identical keystreams becomes $2^{15.99} \times 2^{-23.2} \approx 2^{-7.2}$. With about $2^{7.2} \times 512 = 2^{16.2}$ IVs, identical keystreams can be obtained.

Experiment 7.2. We use 2^{16} random 128-bit keys in the improved attack. For each key, we generate 512 128-bit IVs with the values of the least significant bit of iv[4] and the eight bits of iv[5] choosing all the 512 possible values, while all the other 119 IV bits remain unchanged for each key (but those 119 IV bits are random from key to key). Then we obtain $255 \times 255 = 2^{15.99}$ IV pairs with the required difference. Among these $2^{16} \times 2^{15.99} \approx 2^{32}$ IV pairs, 447 IV pairs result in identical keystreams.

The above experiment shows that with $2^{16} \times 512 = 2^{25}$ selected IVs, 447 IVs result in identical keystreams. It shows that two identical keystreams appear for every $\frac{2^{25}}{447} = 2^{16.2}$ IVs.

For the IV pair with three-byte difference, a similar improvement can also be applied.

Experiment 7.3. We use 2^{16} random 128-bit keys in the improved attack. For each key, we generate 512 128-bit IVs with the values of the most significant bit of iv[4] and the eight bits of iv[5] choosing all the 512 possible values, and the most significant bit of iv[8] is different from the most significant bit of iv[4], while all the other 118 IV bits remain unchanged for each key (but those 118 IV bits are randomly generated for each key). Then we obtain $255 \times 255 = 2^{15.99}$ IV pairs with the required difference. Among these $2^{16} \times 2^{15.99} \approx 2^{32}$ IV pairs, 570 IV pairs result in identical keystreams.

The above experiment shows that with $2^{16} \times 512 = 2^{25}$ selected IVs, 570 IVs result in identical keystreams. It means that two identical keystreams appear for every $\frac{2^{25}}{570} = 2^{15.9}$ IVs.

Remarks. The attacks show that the Py and Pypy are practically insecure. In the application, if the IVs are generated from a counter, or if the IV is short (such as 3 or 4 bytes), then the special IVs (with the differences as illustrated above) appear with high probability, and identical keystreams can be obtained with high probability.

7.4 Key Recovery Attack on Py and Pypy

In this section, we develop a key recovery attack against Py and Pypy by exploiting the collision in the internal state. The key recovery attack consists of two stages: recovering part of the array Y in the IV setup and recovering the key information from Y in the key setup.

7.4.1 Recovering part of the array Y

We use the following IV differences to illustrate the attack (the other IV differences can also be used). Let two IVs iv_1 and iv_2 differ only in two bytes, $iv_1[i] \oplus iv_2[i] = 1$, $iv_1[i+1] \neq iv_2[i+1]$ ($i \geq 1$), and the least significant bit of $iv_1[i]$ be 1. This type of IV pair results in identical keystreams with probability $2^{-23.2}$.

We first recover part of Y from Algorithm A in the IV setup (more information of Y will be recovered from Algorithm B).

Note that the permutation P in Algorithm A is unknown. According to the IV setup algorithm, there is 15 bits of secret information in P, i.e., there are at most 2^{15} possible permutations. During the recovery of Y, we assume that P is known (the effect of the 15-bit secret information in P will be analyzed in Sect. 7.4.2). For iv_m , denote the s at the end of the jth step of Algorithm A as s_j^m , and denote the least and most significant bytes of s_j^m as $s_{j,0}^m$ and $s_{j,3}^m$, respectively. Denote the least and most significant bytes of Y(j) with $Y_{j,0}$ and $Y_{j,3}$, respectively. Note that in Algorithm A, Y remains the same for all the IVs. Denote ξ as a binary random variable with value 0 with probability 0.5. Denote with B(x) a function that gives the least significant byte of x. If the keystreams for iv_1 and iv_2 identical, then from the analysis given in Sect. 7.3.1, we know that $s_{i+1}^1 = s_{i+1}^2$, i.e.,

$$s_i^1 + iv_1[i+1] = s_i^2 + iv_2[i+1].$$
(7.1)

From Algorithm A, we know

$$s_{i} = \text{ROTL32}(s_{i-1} + iv[i] + Y(-3 + i), 8) \oplus P(B(s_{i-1} + iv[i] + Y(-3 + i)))$$
(7.2)

Thus we obtain

$$s_{i,0} = P(B(s_{i-1,0} + iv[i] + Y(-3 + i))) \oplus B(s_{i-1,3} + Y(-3 + i) + \xi_i), \quad (7.3)$$
$$(s_i^1 - s_{i,0}^1) - (s_i^2 - s_{i,0}^2) = (iv_1[i] - iv_2[i]) \ll 8 = 256, \quad (7.4)$$

where ξ_i is caused by the carry bits at the 24th least significant bit position when

iv[i] and Y(-3+i) are introduced, and (7.4) holds with probability $1-2^{-15}$. From (7.1), (7.3) and (7.4), we obtain

$$(P(B(s_{i-1,0}^{1}+iv_{1}[i]+Y_{-3+i,0})) \oplus B(s_{i-1,3}^{1}+Y_{-3+i,3}+\xi_{i,1})) + 256 + iv_{1}[i+1]$$

= $(P(B(s_{i-1,0}^{2}+iv_{2}[i]+Y_{-3+i,0})) \oplus B(s_{i-1,3}^{2}+Y_{-3+i,3}+\xi_{i,2})) + iv_{2}[i+1],$
(7.5)

where $\xi_{i,1} = \xi_{i,2}$ with probability $1 - 2^{-15}$ since the iv[i] has a negligible effect on the value of ξ_1 and ξ_2 . In the following, we use ξ_i to represent $\xi_{i,1}$ and $\xi_{i,2}$.

Denote iv_{θ} as a fixed IV with the first *i* bytes being identical to all the IVs with differences only at iv[i] and iv[i+1]. Thus $s_{i-1,0}^{\theta} = s_{i-1,0}^{1} = s_{i-1,0}^{2}$, and $s_{i-1,3}^{\theta} = s_{i-1,3}^{1} = s_{i-1,3}^{2}$. (7.5) becomes

$$(P(B(s_{i-1,0}^{\theta}+iv_{1}[i]+Y_{-3+i,0})) \oplus B(s_{i-1,3}^{\theta}+Y_{-3+i,3}+\xi_{i})) + 256 + iv_{1}[i+1]$$

= $(P(B(s_{i-1,0}^{\theta}+iv_{2}[i]+Y_{-3+i,0})) \oplus B(s_{i-1,3}^{\theta}+Y_{-3+i,3}+\xi_{i})) + iv_{2}[i+1].$
(7.6)

Using another IV pair different at iv[i] and iv[i+1], and the first *i* bytes being the same as iv_{θ} , another equation (7.6) can be obtained if there is a collision in their internal states. Suppose that several equations (7.6) are available. We consider that the value of ξ_i is independent of iv[i] in the following attack since ξ_i is affected by iv[i] with small probability 2^{-15} . We can recover the values of $B(s_{i-1,0}^{\theta} + Y_{-3+i,0})$ and $B(s_{i-1,3}^{\theta} + Y_{-3+i,3} + \xi_i)$. From the experiment, we find that if there are two equations (7.6), on average the correct values can be recovered together with 5.22 wrong values. If there are three, four, five, six, seven equations (7.6), in average the correct values can be recovered together with 1.29, 0.54, 0.25, 0.12, 0.06 wrong values, respectively. It shows that the values of $B(s_{i-1,0}^{\theta} + Y_{-3+i,0})$ and $B(s_{i-1,3}^{\theta} + Y_{-3+i,3} + \xi_i)$ can be determined with only a few equations (7.6).

After recovering several consecutive $B(s_{i-1,0}^{\theta}+Y_{-3+i,0})$ and $B(s_{i-1,3}^{\theta}+Y_{-3+i,3}+\xi_i)$ $(i \geq 1)$, we proceed to recover part of the information of the array Y. From the values of $B(s_{i-1,0}^{\theta}+Y_{-3+i,0})$, $B(s_{i-1,3}^{\theta}+Y_{-3+i,3}+\xi_i)$ and (7.3), we determine the value of $s_{i,0}^{\theta}$. From the values of $B(s_{i,0}^{\theta}+Y_{-3+i+1,0})$ and $s_{i,0}^{\theta}$, we know the value of $Y_{-3+i+1,0}$.

Generating the equations (7.6). The above attack can only be successful if we can find several equations (7.6) with the same $s_{i-1,0}^{\theta}$ and $s_{i-1,3}^{\theta}$. In the following, we illustrate how to obtain these equations for $2 \leq i \leq ivsizeb - 3$. At the beginning of the attack, we set a fixed iv_{θ} . For all the IVs different at only iv[i]and iv[i+1], we require that their first *i* bytes are identical to that of iv_{θ} . Let the least significant bit of iv[i] and the 8 bits of iv[i+1] choose all the 512 values, and the other 119 bits remain unchanged, then we obtain a 255 × 255 $\approx 2^{16}$ desired IV pairs. We call these 512 IVs a desired IV group. According to Experiment 7.2, this type of IV pair results in identical keystreams with probability $2^{-23.2}$, we thus obtain $\frac{2^{-23.2}}{2^{16}} = 2^{-7.2}$ identical keystream pairs from one desired IV group. It means that we can obtain $2^{-7.2}$ equations (7.1) from one desired IV group. We modify the values of the 7 most significant bits of $iv_1[i]$ and $iv_2[i]$, and 3 bits of $iv_1[i+2]$ and $iv_2[i+2]$, then we obtain $2^7 \times 2^3 = 2^{10}$ desired IV groups. From these desired IV groups, we obtain $2^{10} \times 2^{-7.2} = 7$ equations (7.1). There are $2^7 \times 2^3 \times 2^9 = 2^{19}$ IVs being used in the attack. To find all the $s_{i,0}$ for $2 \leq i \leq ivsizeb - 3$, we need (ivsizeb - 4) $\times 2^{19}$ IVs in the attack.

We are able to recover $s_{i,0}^{\theta}$ for $2 \leq i \leq ivsizeb - 3$, which implies that we can recover the values of $Y_{-3+i,0}$ for $3 \leq i \leq ivsizeb - 3$. Then we proceed to recover more information of Y by considering Algorithm B. Applying an attack similar to the above attack and reusing the IVs, we can recover the values of $Y_{256-i,0}$ for $3 \leq i \leq ivsizeb - 3$.

Thus with $(ivsizeb - 4) \times 2^{19}$ IVs, we are able to recover $2 \times (ivsizeb - 6)$

bytes of Y: $Y_{-3+i,0}$ and $Y_{256-i,0}$ for $3 \le i \le ivsizeb - 3$.

7.4.2 Recovering the key

In the above analysis, we recovered the values of $Y_{-3+i,0}$ and $Y_{256-i,0}$ for $3 \leq i \leq ivsizeb-3$ by exploiting the difference elimination in s. Next, we will recover the 15-bit secret information in P by exploiting the difference elimination in EIV. Denote s_i^{θ} in Algorithm A and B as $s_i^{A,\theta}$ and $s_i^{B,\theta}$, respectively. Denote $EIV_1[i]$ at the end of Algorithm A and B as $EIV_1^A[i]$ and $EIV_1^B[i]$, respectively. For two IVs differing in only iv[i] and iv[i+1] and generating identical keystreams, $EIV_1^A[i]$, $EIV_2^A[i]$, $EIV_2^B[i]$ and $EIV_2^B[i]$ are computed as:

$$EIV_1^A[i] = P(B(s_{i-1,0}^{A,\theta} + iv_1[i] + Y_{-3+i,0}))$$
(7.7)

$$EIV_{2}^{A}[i] = P(B(s_{i-1,0}^{A,\theta} + iv_{2}[i] + Y_{-3+i,0}))$$
(7.8)

$$EIV_1^B[i] = EIV_1^A[i] + P(B(s_{i-1,0}^{B,\theta} + iv_1[i] + Y_{256-i,0}))$$
(7.9)

$$EIV_2^B[i] = EIV_2^A[i] + P(B(s_{i-1,0}^{B,\theta} + iv_2[i] + Y_{256-i,0}))$$
(7.10)

Since the two keystreams are identical, it is required that

$$EIV_1^B[i] = EIV_2^B[i]. (7.11)$$

Note that the values of $B(s_{i-1,0}^{A,\theta}+Y_{-3+i,0})$ and $B(s_{i-1,0}^{B,\theta}+Y_{256-i,0})$ are determined when we recover part of Y from Algorithm A and Algorithm B, respectively. Eight bits of information on P is revealed from (7.7), (7.8), (7.9), (7.10) and (7.11). In Sect. 7.4.1, there are about 7 pairs of IVs resulting in identical keystreams for each value of i. Thus P can be recovered completely.

We proceed to recover the key information. We consider the last part of the key schedule:

```
for(i=YMININD, j=0; i<=YMAXIND; i++)
{
    s = s + key[j];
    s0 = internal_permutation[s&0xFF];
    Y(i) = s = ROTL32(s, 8) ^ (u32)s0;
    j = (j+1) mod keysizeb;
}</pre>
```

We call the above algorithm Algorithm C. From Algorithm C, we obtain the following relation:

$$B(Y_{-3+i,0} + key[i + 1 \mod keysizeb] + \xi'_i) \oplus P'(B(Y_{-3+i+3,0} + key[i + 4 \mod keysizeb])) = Y_{-3+i+4,0},$$
(7.12)

where P' indicates the 'internal_permutation', ξ'_i indicates the carry bit noise introduced by key[i+2] and key[i+3]; it is computed as $\xi'_i \approx (key[i+2] + Y_{-3+i+1,0}) \gg 8$. The value of the bit ξ'_i is 0 with probability about 0.5.

Once the values of $Y_{-3+i,0}$ $(3 \le i \le ivsizeb-3)$ are known, we find a relation (7.12) linking $key[i + 1 \mod keysizeb]$ and $key[i + 4 \mod keysizeb]$ for $3 \le i \le ivsizeb - 7$. Each relation leaks at least 7 bits of $key[i + 1 \mod keysizeb]$ and $key[i + 4 \mod keysizeb]$. The values of $Y_{256-i,0}$ $(3 \le i \le ivsizeb - 3)$ are also known, thus we can find a relation (7.12) linking $key[i + 1 \mod keysizeb]$ and $key[i + 4 \mod keysizeb]$ for $262 - ivsizeb \le i \le 252$. Thus there are $2 \times (ivsizeb - 9)$ relations (7.12) linking the key bytes.

For the 16-byte key and 16-byte IV, 14 relations (7.12) can be obtained: 7 relations linking key[i] and key[i+3] for $4 \le i \le 10$, and another 7 relations (7.12) linking key[i] and $key[i+3 \mod 16]$ for $7 \le i \le 13$. There are 13 key bytes in these 14 relations (7.12). Note that the randomness of ξ'_i does not affect the overall attack (once we guess the values of key[4], key[5] and key[6], then we obtain the other key bytes key[j] ($7 \le j \le 15$), key[0], and all the ξ'_j ($3 \le j \le 9$ and $247 \le j \le 249$). Thus these 14 relations are sufficient to recover the 13 key bytes. The effective key size is reduced to 3 bytes and these three bytes can be found easily with brute force search.

For the 32-byte key and 32-byte IV, 46 relations (7.12) can be obtained: 23 relations linking key[i] and key[i+3] for $4 \le i \le 26$, and another 23 relations (7.12) linking key[i] and $key[i+3 \mod 32]$ for $7 \le i \le 29$. There are 29 key bytes in these 46 relations. The effective key size is again reduced to 3 bytes.

7.5 The Security of Py6

Py6 is a variant of Py with reduced internal state size. The array P is a permutation with only 64 elements, and the array Y has 68 entries. Py6 was proposed to achieve fast initialization, but it is weaker than Py. Paul and Preneel has developed distinguishing attack against Py6 with data complexity $2^{68.6}$ [110]. In the following, we show that identical keystreams are generated from Py6 with high probability. There is no detailed description of the key and IV setups of Py6. Thus we use the source code of Py6 submitted to eSTREAM as reference. In our experiment, the following IV differences are used: $iv_1[i] - iv_2[i] = 32$, $iv_1[i+1] \neq iv_2[i+1]$, $iv_1[i+1] \gg 6 = iv_2[i+1] \gg 6$, and $iv_2[i+5] - iv_1[i+5] = 8$ $(i \geq 2)$. After testing 2^{30} pairs with the original Py6 source code, we found that identical keystreams appear with probability $2^{-11.45}$. This probability is much larger than the probability 2^{-23} for Py and Pypy. It shows that Py6 is much weaker than Py and Pypy.

7.6 Conclusion

In this chapter, we developed practical differential attacks against Py, Py6 and Pypy: the identical keystreams appear with high probability, and the key information can be recovered when the IV size is more than 9 bytes. To resist the attacks given in this chapter, we suggest that the IV setup be performed in an invertible way.

Several ciphers in the eSTREAM competition have been broken due to the flaws in their IV setups: DECIM [130], WG [131], LEX [131], Py, Pypy and VEST [77]. We should pay great attention to the design of the stream cipher IV setup.

84 CHAPTER 7. DIFFERENTIAL ATTACKS ON PY, PY6 AND PYPY

Chapter 8

The Stream Cipher HC-256

Abstract. The stream cipher HC-256 is proposed in this chapter. It generates keystream from a 256-bit secret key and a 256-bit initialization vector. HC-256 consists of two secret tables, each one with 1024 32-bit elements. The two tables are used as S-Box alternatively. At each step one element of a table is updated and one 32-bit output is generated. The encryption speed of the C implementation of HC-256 is about 1.9 bits per clock cycle (4.2 clock cycles per byte) on the Intel Pentium 4 processor.

8.1 Introduction

Stream ciphers are used for shared-key encryption. The modern software efficient stream ciphers can run 4-to-5 times faster than block ciphers. However, very few efficient and secure stream ciphers have been published. Even the most widely used stream cipher RC4 [114] has several weaknesses [60, 82, 95, 51, 50, 86, 94]. In the recent NESSIE project all the six stream cipher submissions cannot meet the stringent security requirements [105]. In this chapter we aim to design a very simple, secure, software-efficient and freely-available stream cipher.

HC-256 is the stream cipher we propose in this chapter. It consists of two secret tables, each one with 1024 32-bit elements. At each step we update one element of a table with non-linear feedback function. Every 2048 steps all the elements of the two tables are updated. At each step, HC-256 generates one 32-bit output using the 32-bit-to-32-bit mapping similar to that being used in Blowfish [116]. Then the linear masking is applied before the output is generated.

In the design of HC-256, we take into consideration the superscalar feature of modern (and future) microprocessors. Without compromising the security, we try to reduce the dependency between operations. The dependency between the steps is reduced so that three consecutive steps can be computed in parallel. At each step, three parallel additions are used in the feedback function and three additions are used to combine the four table lookup outputs instead of the addition-xor-addition being used in Blowfish (a similar idea has been suggested by Schneier and Whiting to use three xors to combine those four terms [117]).

With the high degree of parallelism, HC-256 runs very efficiently on the modern processor. We implemented HC-256 in C and tested its performance on the Pentium 4 processor. The encryption speed of HC-256 reaches 1.93 bit/cycle.

This chapter is organized as follows. We introduce HC-256 in Sect. 8.2. The security of HC-256 is analyzed in Sect. 8.3. Section 8.4 discusses the implementation and performance of HC-256. Section 8.5 concludes this chapter.

8.2 Stream Cipher HC-256

In this section, we describe the stream cipher HC-256. From a 256-bit key and a 256-bit initialization vector, it generates keystream with a length up to 2^{128} bits.

8.2.1 Operations, variables and functions

The following operations are used in HC-256:

- + : x + y means $x + y \mod 2^{32}$, where $0 \le x < 2^{32}$ and $0 \le y < 2^{32}$
- \exists : $x \exists y \text{ means } x y \mod 1024$
- \oplus : bit-wise exclusive OR
- : concatenation
- \gg : right shift operator. $x \gg n$ means x being right shifted over n bit positions.
- \ll : left shift operator. $x \ll n$ means x being left shifted over n bit positions.
- \implies : right rotation operator. $x \implies n$ means x being rotated to the right over n bit positions.

Two tables P and Q are used in HC-256. The key and the initialization vector of HC-256 are denoted as K and IV. We denote the keystream being generated as s.

- P : a table with 1024 32-bit elements. Each element is denoted as P[i] with $0 \leq i \leq 1023.$
- Q : a table with 1024 32-bit elements. Each element is denoted as $Q[i] \text{ with } 0 \leq i \leq 1023.$
- K : the 256-bit key of HC-256.
- IV : the 256-bit initialization vector of HC-256.
- s : the keystream being generated from HC-256. The 32-bit output of the *i*th step is denoted as s_i . Then $s = s_0 ||s_1||s_2|| \cdots$

There are six functions being used in HC-256. $f_1(x)$ and $f_2(x)$ are the same as the $\sigma_0^{\{256\}}(x)$ and $\sigma_1^{\{256\}}(x)$ being used in the message schedule of SHA-256 [100]. For $g_1(x)$ and $h_1(x)$, the table Q is used as S-box. For $g_2(x)$ and $h_2(x)$, the table P is used as S-box.

$$\begin{array}{rcl} f_1(x) &=& (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3) \\ f_2(x) &=& (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10) \\ g_1(x,y) &=& ((x \ggg 10) \oplus (y \ggg 23)) + Q[(x \oplus y) \bmod 1024] \\ g_2(x,y) &=& ((x \ggg 10) \oplus (y \ggg 23)) + P[(x \oplus y) \bmod 1024] \\ h_1(x) &=& Q[x_0] + Q[256 + x_1] + Q[512 + x_2] + Q[768 + x_3] \\ h_2(x) &=& P[x_0] + P[256 + x_1] + P[512 + x_2] + P[768 + x_3] , \end{array}$$

where $x = x_3 ||x_2||x_1||x_0$, x is a 32-bit word, x_0 , x_1 , x_2 and x_3 are four bytes. x_3 and x_0 denote the most significant byte and the least significant byte of x, respectively.

8.2.2 Initialization process (key and IV setup)

The initialization process of HC-256 consists of expanding the key and initialization vector into P and Q (similar to the message setup in SHA-256) and running the cipher 4096 steps without generating output.

1. Let $K = K_0 ||K_1|| \cdots ||K_7$ and $IV = IV_0 ||IV_1|| \cdots ||IV_7$, where each K_i and IV_i denotes a 32-bit number. The key and IV are expanded into an array W_i ($0 \le i \le 2559$) as:

$$W_{i} = \begin{cases} K_{i} & 0 \le i \le 7\\ IV_{i-8} & 8 \le i \le 15\\ f_{2}(W_{i-2}) + W_{i-7} + f_{1}(W_{i-15}) + W_{i-16} + i & 16 \le i \le 2559 \end{cases}$$

2. Update the tables P and Q with the array W.

 $P[i] = W_{i+512} \quad \text{for } 0 \le i \le 1023$ $Q[i] = W_{i+1536} \quad \text{for } 0 \le i \le 1023$

3. Run the cipher (the keystream generation algorithm in Sect. 8.2.3) 4096 steps without generating output.

The initialization process completes and the cipher is ready to generate keystream.

8.2.3 The keystream generation algorithm

At each step, one element of a table is updated and one 32-bit output is generated. An S-box is used to generate only 1024 outputs, then it is updated in the next 1024 steps. The keystream generation process of HC-256 is given below (" \boxminus " denotes " \neg " modulo 1024, s_i denotes the output of the *i*-th step).

```
i = 0;
repeat until enough keystream bits are generated.
{
       j = i \mod 1024;
       if (i \mod 2048) < 1024
       {
             \begin{split} P[j] &= P[j] + P[j \boxminus 10] + g_1(P[j \boxminus 3], P[j \boxminus 1023]);\\ s_i &= h_1(P[j \boxminus 12]) \oplus P[j]; \end{split}
       }
       else
       {
             Q[j] = Q[j] + Q[j \boxminus 10] + g_2(Q[j \boxminus 3], Q[j \boxminus 1023]);
             s_i = h_2(Q[j \boxminus 12]) \oplus Q[j];
       }
       end-if
       i = i + 1;
}
end-repeat
```

8.2.4 Encryption and decryption

The keystream is XORed with the message for encryption. The decryption is to XOR the keystream with the ciphertext. The test vectors of HC-256 are given in Appendix B.1.
8.3 Security Analysis of HC-256

We start with a brief review of the attacks on stream ciphers. Many stream ciphers are based on the linear feedback shift registers (LFSRs) and a number of correlation attacks, such as [120, 121, 89, 57, 93, 33, 75], were developed to analyze them. Later Golić [58] devised the linear cryptanalysis of stream ciphers. That technique could be applied to a wide range of stream ciphers. Recently Coppersmith, Halevi and Jutla [36] developed distinguishing attacks (the linear attack and low diffusion attack) on stream ciphers with linear masking.

Fast correlation attacks cannot be applied to HC-256 because HC-256 uses non-linear feedback functions to update the two tables P and Q. The output function of HC-256 uses the 32-bit-to-32-bit mapping similar to that being used in Blowfish. The analysis on Blowfish shows that it is extremely difficult to apply linear cryptanalysis [88] to the large secret S-box. The large secret S-box of HC-256 is updated during the keystream generation process and it is almost impossible to develop linear relations linking the input and output bits of the S-box. Vaudenay has found some differential weakness of a randomly generated large S-box [124]. But it is very difficult to launch differential cryptanalysis [27] against HC-256 since it is a synchronous stream cipher for which the keystream generation is independent of the message.

In this section, we will analyze the security of the secret key, the randomness of the keystream, and the security of the initialization process.

8.3.1 Period

The 65547-bit state of HC-256 ensures that the period of the keystream is extremely large. But the exact period of HC-256 is difficult to predict. The average period of the keystream is estimated to be about 2^{65546} (if we assume that the invertible next-state function of HC-256 is random). The large number of states also completely eliminates the threat of time-memory tradeoff attack on stream ciphers [7, 59].

8.3.2 The security of the key

We begin with the study of a modified version of HC-256 (without linear masking). Our analysis shows that even for this weak version of HC-256, it is impossible to recover the secret key faster than exhaustive key search. The reason is that the keystream is generated from a highly non-linear function $(h_1(x) \text{ or } h_2(x))$, so the keystream leaks a very small amount of information at each step. Recovering P and Q requires partial information leaked from a lot of steps. Because the tables are updated in a highly non-linear way, it is difficult to retrieve the information of P and Q from the leaked information.

HC-256 with no linear masking.

For HC-256 with no linear masking, the output at the *i*th step is generated as $s_i =$ $h_1(P[i \boxminus 12])$ or $s_i = h_2(Q[i \boxminus 12])$. If two outputs generated from the same S-box are equal, then very likely those two inputs to the S-box are equal. According to the analysis on the randomness of the outputs of $h_1(x)$ and $h_2(x)$ given in Sect. 8.3.3, $s_{2048 \times \alpha + i} = s_{2048 \times \alpha + j}$ ($0 \le i < j < 1024$) with probability about 2^{-31} . If $s_{2048 \times \alpha + i} = s_{2048 \times \alpha + j}$, then at the $(2048 \times \alpha + j)$ -th step, $P[i \boxminus 12] =$ $P[j \boxminus 12]$ with probability about 0.5 (31-bit information of the table P is leaked). We note that for every 1024 steps in the range $(2048 \times \alpha, 2048 \times \alpha + 1024)$, the same S-box is used in $h_1(x)$. The probability that there are two equal outputs is $\binom{1024}{2} \times 2^{-31} \approx 2^{-12}$. On average each output leaks $\frac{2^{-12} \times 31}{1024} \approx 2^{-17}$ information bits of the table P. To recover P, we need to analyze at least $\frac{1024 \times 32}{2^{-17}} \approx 2^{32}$ outputs. Recovering P from those 2^{32} outputs involves very complicated nonlinear equations and solving them is computationally infeasible. Recovering Q is as difficult as recovering P. We note that the table Q is used as S-box to update P, and vice versa. P and Q interact in such a complicated way and recovering them from the keystream cannot be faster than exhaustive key search.

HC-256

The analysis above shows that the secret key of HC-256 with no linear masking is secure. With the linear masking, the information leakage is greatly reduced and it would be even more difficult to recover the secret key from the keystream. We thus conclude that the key of HC-256 cannot be recovered faster than exhaustive key search.

8.3.3 Randomness of the keystream

In this subsection, we investigate the randomness of the keystream of HC-256. As large, secret and frequently updated S-boxes are used in the cipher, the most efficient attack is to analyze the randomness of the overall 32-bit words. Under this guideline, we developed some attacks against HC-256 with no linear masking. Then we show that the linear masking eliminates those threats.

Keystream of HC-256 with no linear masking

The goal of the attacks on HC-256 with no linear masking is to investigate the security weaknesses in the output and feedback functions. We developed two attacks against HC-256 with no linear masking.

Weakness of $h_1(x)$ and $h_2(x)$. For HC-256 with no linear masking, the output is generated as $s_i = h_1(P[i \boxminus 12])$ or $s_i = h_2(Q[i \boxminus 12])$. Because there is no difference between the analysis of $h_1(x)$ and $h_2(x)$, we use h(x) to refer to $h_1(x)$ and $h_2(x)$ here. Assume that h(x) is a 32-bit-to-32-bit S-box H(x) with randomly generated secret elements and the inputs to H are randomly generated. Because the elements of the H(x) are randomly generated, the output of H(x)is not uniformly distributed. If many outputs are generated from H(x), some values in the range $[0, 2^{32})$ never appear and some appear with probability larger than 2^{-32} . Then it is straightforward to distinguish the outputs from random. However each H(x) in HC-256 is used to generate only 1024 outputs, then it gets updated. The direct computation of the distribution of the outputs of H(x)from those 1024 outputs cannot be successful. Instead, we consider the collision between the outputs of H(x).

Theorem 8.3.1 Let H be an m-bit-to-n-bit S-box and all those n-bit elements are randomly generated, where $m \ge n$ and n is a large integer. Let x_1 and x_2 be two m-bit random inputs to H. Then $H(x_1) = H(x_2)$ with probability about $2^{-n} + 2^{-m}$.

Proof. If $x_1 = x_2$, then $H(x_1) = H(x_2)$. If $x_1 \neq x_2$, then $H(x_1) = H(x_2)$ with probability 2^{-n} . $x_1 = x_2$ with probability 2^{-m} and $x_1 \neq x_2$ with probability $1 - 2^{-m}$. The probability that $H(x_1) = H(x_2)$ is $2^{-m} + (1 - 2^{-m}) \times 2^{-n} \approx 2^{-n} + 2^{-m}$.

Attack 1. According to Theorem 8.3.1, for the 32-bit-to-32-bit S-box H, the collision rate of the outputs is $2^{-32} + 2^{-32} = 2^{-31}$. With 2^{35} pairs of $(H(x_1), H(x_2))$, we can distinguish the output from random with success rate 0.761. (The success rate can be improved to 0.996 with 2^{36} pairs.) Note that only 1024 outputs are generated from the same S-box H, so 2^{26} outputs are needed to distinguish the keystream of HC-256 with no linear masking.

Experiment 8.1. To compute the collision rate of the outputs of HC-256 (with no linear masking), we generated 2^{39} outputs (2^{48} pairs). The collision rate is $2^{-31} - 2^{-40.09}$. The experiment confirms that the collision rate of the outputs of h(x) is very close to 2^{-31} , and approximating h(x) with a randomly generated S-box has negligible effect on the attack.

Remarks. The distinguishing attack above can be slightly improved if we consider the differential attack on Blowfish. Vaudenay [124] has pointed out that the collision in a randomly generated S-box in Blowfish can be applied to distinguish the outputs of Blowfish with a reduced number of rounds (8 rounds). The basic

idea of Vaudenay's differential attack is that if Q[i] = Q[j] for $0 \le i, j < 256$, $i \ne j$, then for $a_0 \oplus a'_0 = i \oplus j$, $h_1(a_3||a_2||a_1||a_0) = h_1(a_3||a_2||a_1||a'_0)$ with probability 2^{-7} , where each a_i denotes an 8-bit number. We can detect the collision in the S-box with success rate 0.5 since that S-box Q is used as inputs to $h_2(x)$ to produce 1024 outputs. If Q[i] = Q[j] for $256\alpha \le i, j < 256\alpha + 256, 0 \le \alpha < 4$, $i \ne j$, and x_1 and x_2 are two random inputs (note that we cannot introduce or identify inputs with particular difference to h(x)), then the probability that $h_1(x_1) = h_1(x_2)$ becomes $2^{-31} + 2^{-32}$. However the chance that there is one useful collision in the S-box is only $\frac{\binom{256}{2}\times 4}{2^{322}} = 2^{-15}$. The average collision rate becomes $2^{-15} \times (2^{-31} + 2^{-32}) + (1 - 2^{-15}) \times 2^{-31} = 2^{-31} + 2^{-47}$. The increase in collision rate is so small that the collision in the S-box has a negligible effect on this attack.

Weakness of the feedback function. The table P is updated with the nonlinear feedback function $P[i \mod 1024] = P[i \mod 1024] + P[i \boxminus 10] + g_1(P[i \boxminus 3], P[i \boxminus 1023])$. The following attack tries to distinguish the keystream by exploiting this relation.

Attack 2. Assume that the h(x) is a one-to-one mapping. Consider two groups of outputs $(s_i, s_{i-3}, s_{i-10}, s_{i-2047}, s_{i-2048})$ and $(s_j, s_{j-3}, s_{j-10}, s_{j-2047}, s_{j-2048})$. If $i \neq j$ and $1024 \times \alpha + 10 \leq i, j < 1024 \times \alpha + 1023$, they are equal with probability about 2^{-128} . The collision rate is 2^{-160} if the outputs are truly random. 2^{-128} is much larger than 2^{-160} , so the keystream can be distinguished from random with about 2^{128} pairs of such five-tuple groups of outputs. Note that the S-box is updated every 1024 steps, hence 2^{119} outputs are needed in the attack.

The two attacks given above show that the HC-256 with no linear masking does not generate secure keystream.

Keystream of HC-256

With the linear masking being applied, it is no longer possible to exploit those two weaknesses separately and the attacks given above cannot be applied directly. We need to remove the linear masking first. We recall that at the *i*th step, if $(i \mod 2048) < 1024$, the table P is updated as

 $P[i \mod 1024] = P[i \mod 1024] + P[i \boxminus 10] + g_1(P[i \boxminus 3], P[i \boxminus 1023]).$

We know that $s_i = h_1(P[i \boxminus 12]) \oplus P[i \mod 1024]$. For $10 \le (i \mod 2048) < 1023$, this feedback function can be written alternatively as

$$s_{i} \oplus h_{1}(z_{i}) = (s_{i-2048} \oplus h'_{1}(z_{i-2048})) + (s_{i-10} \oplus h_{1}(z_{i-10})) + g_{1}(s_{i-3} \oplus h_{1}(z_{i-3}), s_{i-2047} \oplus h'_{1}(z_{i-2047})),$$

$$(8.1)$$

where $h_1(x)$ and $h'_1(x)$ indicate two different functions since they are related to different S-boxes; z_j denotes the $P[j \boxminus 12]$ at the *j*-th step. The linear masking is removed successfully in (8.1). However, it is very difficult to apply (8.1) directly to distinguish the keystream. To simplify the analysis, we attack a weak version of (8.1). We replace all the '+' in the feedback function with ' \oplus ' and write (8.1) as

$$s_{i} \oplus s_{i-2048} \oplus s_{i-10} \oplus (s_{i-3} \implies 10) \oplus (s_{i-2047} \implies 23)$$

= $h_{1}(z_{i}) \oplus h'_{1}(z_{i-2048})) \oplus h_{1}(z_{i-10})) \oplus (h_{1}(z_{i-3}) \implies 10) \oplus$
 $\oplus (h'_{1}(z_{i-2047}) \implies 23) \oplus Q[r_{i}],$ (8.2)

where $r_i = (s_{i-3} \oplus h_1(z_{i-3}) \oplus s_{i-2047} \oplus h'_1(z_{i-2047})) \mod 1024$. Because of the random nature of $h_1(x)$ and Q, the right hand side of (8.2) is not uniformly distributed. But each S-box is used in only 1024 steps, these 1024 outputs are not sufficient to compute the distribution of $s_i \oplus s_{i-2048} \oplus s_{i-10} \oplus (s_{i-3} \gg 10) \oplus (s_{i-2047} \gg 23)$. Instead we need to study the collision rate. The effective way is to eliminate the term $h_1(z_i)$ before analyzing the collision rate.

Replace the i with i+10. For $10 \leq i \mbox{ mod } 2048 < 1013, (8.2)$ can be written as

$$s_{i+10} \oplus s_{i-2038} \oplus s_i \oplus (s_{i+7} \implies 10) \oplus (s_{i-2037} \implies 23)$$

= $h_1(z_{i+10}) \oplus h'_1(z_{i-2038})) \oplus h_1(z_i) \oplus (h_1(z_{i+7}) \implies 10) \oplus$
 $\oplus (h'_1(z_{i-2037}) \implies 23) \oplus Q[r_{i+10}].$ (8.3)

For the left-hand sides of (8.2) and (8.3) to be equal, i.e., for the following equation

$$s_{i} \oplus s_{i-2048} \oplus s_{i-10} \oplus (s_{i-3} \gg 10) \oplus (s_{i-2047} \gg 23) = s_{i+10} \oplus s_{i-2038} \oplus s_{i} \oplus (s_{i+7} \gg 10) \oplus (s_{i-2037} \gg 23)$$
(8.4)

to hold, we require that (after eliminating the term $h_1(z_i)$)

$$h_{1}(z_{i-10}) \oplus h'_{1}(z_{i-2048}) \oplus (h_{1}(z_{i-3}) \Longrightarrow 10) \oplus (h'_{1}(z_{i-2047}) \Longrightarrow 23) \oplus Q[r_{i}] = h_{1}(z_{i+10}) \oplus h'_{1}(z_{i-2038}) \oplus (h_{1}(z_{i+7}) \ggg 10) \oplus (h'_{1}(z_{i-2037}) \ggg 23) \oplus Q[r_{i+10}].$$

$$(8.5)$$

For $22 \leq i \mod 2048 < 1013$, we note that $z_{i-10} = z_i \oplus z_{i-2048} \oplus (z_{i-3} \gg 10) \oplus (z_{i-2047} \gg 23)$, and $z_{i+10} = z_i \oplus z_{i-2038} \oplus (z_{i+7} \gg 10) \oplus (z_{i-2037} \gg 23)$. Approximate (8.5) as

$$H(x_1) = H(x_2),$$
 (8.6)

where H denotes a random secret 106-bit-to-32-bit S-box, x_1 and x_2 are two 106bit random inputs, $x_1 = z_{i-3} ||z_{i-2047}||z_{i-2048}||r_i$ and $x_2 = z_{i+7} ||z_{i-2037}||z_{i-2038}||r_{i+10}$. (The effect of z_i is included in H.) According to Theorem 8.3.1, (8.6) holds with probability $2^{-32} + 2^{-106}$. So (8.4) holds with probability $2^{-32} + 2^{-106}$. We approximate the binomial distribution with the normal distribution. The mean $\mu = Np$ and the standard deviation $\sigma = \sqrt{Np(1-p)}$, where N is the total number of equations (8.4), and $p = 2^{-32} + 2^{-106}$. For random signal, $p' = 2^{-32}$, the mean $\mu' = Np'$ and the standard deviation $\sigma' = \sqrt{Np'(1-p')}$. If $|u - u'| > 2(\sigma + \sigma')$, i.e. $N > 2^{184}$, the output of the cipher can be distinguished from random with success rate 0.9772.

After verifying the validity of 2^{184} equations (8.4), we can successfully distinguish the keystream from random. We note that the S-box is updated every 1024 steps, so only about 2^{10} equations (8.4) can be obtained from 1024 steps in the range $1024 \times \alpha \leq i < 1024 \times \alpha + 1024$. To distinguish the keystream from random, 2^{184} outputs are needed in the attack.

The attack above can be improved by exploiting the relation $r_i = (s_{i-3} \oplus h_1(z_{i-3}) \oplus s_{i-2047} \oplus h'_1(z_{i-2047})) \mod 1024$. If $(s_{i-3} \oplus s_{i-2047}) \mod 1024 = (s_{i+7} \oplus s_{i-2037}) \mod 1024$, then (8.6) holds with probability $2^{-32} + 2^{-96}$ and 2^{164} equations (8.4) are needed in the attack. Note that only about one equation (8.4) can now be obtained from 1024 steps in the range $1024 \times \alpha \leq i < 1024 \times \alpha + 1024$. To distinguish the keystream from random, 2^{174} outputs are needed in the attack.

We note that the attack above can only be applied to HC-256 with all the '+' in the feedback function being replaced with ' \oplus '. To distinguish the keystream of HC-256, more than 2^{174} outputs are needed. So we conclude that it is impossible to distinguish a 2^{128} -bit keystream of HC-256 from random.

8.3.4 Security of the initialization process (key/IV setup)

The initialization process of the HC-256 consists of two stages, as given in Sect. 8.2.2. We expand the key and IV into P and Q. At this stage, every bit of the key/IV affects all the bits of the two tables and any difference in the related keys/IVs results in uncontrollable differences in P and Q. Then we run the cipher 4096 steps without generating output so that the P and Q become more random. After the initialization process, we expect that any difference in the keys/IVs would not result in a biased keystream.

8.4 Implementation and Performance of HC-256

The direct C implementation of the encryption algorithm given in Sect 8.2.3 runs at about 0.6 bit/cycle on the Pentium 4 processor. The program size is very small

but the speed is only about 1.5 times that of AES [43]. At each step in the direct implementation, we need to compute $(i \mod 2048)$, $i \boxminus 3$, $i \boxminus 10$ and $i \boxminus 1023$. And at each step there is a branch decision based on the value of $(i \mod 2048)$. These operations affect greatly the encryption speed. The optimization process reduces the amount of these operations.

8.4.1 The optimized implementation of HC-256

This subsection describes the optimized C implementation of HC-256. In the optimized code, loop unrolling is used and only one branch decision is made for every 16 steps. The experiment shows that the branch decision in the optimized code affects the encryption speed by less than one percent.

There are several fast implementations of the feedback functions of P and Q. We use the implementation given in Appendix B in [128] because it achieves the best consistency on different platforms. The details of that implementation are given below. The feedback function of P is given as

 $P[i \mod 1024] = P[i \mod 1024] + P[i \boxminus 10] + g_1(P[i \boxminus 3], P[i \boxminus 1023]).$

A register X containing 16 elements is introduced for P. If $(i \mod 2048) < 1024$ and $i \mod 16 = 0$, then at the begining of the *i*th step, $X[j] = P[(i - 16 + j) \mod 1024]$ for $j = 0, 1, \dots 15$, i.e., X contains the values of $P[i \boxminus 16], P[i \boxminus 15], \dots, P[i \boxminus 1]$. In the 16 steps starting from the *i*th step, P and X are updated as

Note that at the *i*th step, two elements of $P[i \boxminus 10]$ and $P[i \boxminus 3]$ can be obtained directly from X. Also for the output function $s_i = h_1(P[i \boxminus 12]) \oplus P[i \mod 1024]$,

the $P[i \boxminus 12]$ can be obtained from X. In this implementation, there is no need to compute $i \boxminus 3$, $i \boxminus 10$ and $i \boxminus 12$.

A register Y with 16 elements is used in the implementation of the feedback function of Q in the same way as that given above.

To reduce the memory requirement and the program size, the initialization process implemented in Appendix B in [128] is not as straightforward as that given in Sect. 8.2.2. To reduce the memory requirement, we do not implement the array W in the program. Instead we implement the key and IV expansion on P and Q directly. To reduce the program size, we implement the feedback functions of those 4096 steps without involving X and Y.

8.4.2 Performance of HC-256

Encryption Speed. We use the C codes given in Appendix B and Appendix C in [128] to measure the encryption speed. The processor used in the test is Pentium 4 (2.4 GHz, 8 KB Level 1 data cache, 512 KB Level 2 cache, no hyperthreading). The speed is measured by repeatedly encrypting the same 512-bit buffer for 2^{26} times (The buffer is defined as 'static unsigned long DATA[16]' in Appendix C in [128]). The encryption speed is given in Table 8.1.

The C implementation of HC-256 is faster than the C implementations of almost all the other stream ciphers. (However different designers may have made different efforts to optimize their codes, so the speed comparison is not absolutely accurate.) SEAL [112] is a software-efficient cipher and its C implementation runs at the speed of about 1.6 bit/cycle on Pentium III processor. Scream [35] runs at about the same speed as SEAL. The C implementation of SNOW2.0 [47] runs at about 1.67 bit/cycle on Pentium 4 processor. TURING [113] runs at about 1.3 bit/cycle on the Pentium III mobile processor. The C implementation of MUGI [125] runs at about 0.45 bit/cycle on the Pentium III processor. The encryption

| Operating | Compiler | Optimization | Speed |
|------------------|---------------------------|--------------|-------------|
| System | | option | (bit/cycle) |
| Windows XP | Intel C++ Compiler 7.1 | -O3 | 1.93 |
| (SP1) | | | |
| | Microsoft Visual $C++6.0$ | -Release | 1.81 |
| | Professional (SP5) | | |
| Red Hat Linux 9 | Intel C++ Compiler 7.1 | -03 | 1.92 |
| (Linux 2.4.20-8) | gcc 3.2.2 | -03 | 1.83 |

Table 8.1: The speed of the C implementation of HC-256 on a Pentium 4

speed of Rabbit [30] is about 2.16 bit/cycle on Pentium III processor, but it is programmed in Assembly language inline in C.

Remarks. In HC-256, there is a dependency between the feedback and output functions since the $P[i \mod 1024]$ (or $Q[i \mod 1024]$) being updated at the *i*th step is immediately used as linear masking. This dependency reduces the speed of HC-256 by about 3%. We do not remove this dependency from the design of HC-256 for security reason. Our analysis shows that each term being used as linear masking should not have been used in an S-box in the previous steps, otherwise the linear masking could be removed much easier. In our optimized implementation, we do not deal with this dependency because its effect on the encryption speed is very limited on the Pentium 4 processor.

Initialization Process. The key setup of HC-256 requires about 74,000 clock cycles (measured by repeating the setup process 2^{16} times on the Pentium 4 processor with Intel C++ compiler 7.1). This amount of computation is more than that required by most of the other stream ciphers (for example, the initialization process of Scream takes 27,500 clock cycles). The reason is that two large S-boxes are used in HC-256. To eliminate the threat of related key/IV attack, the tables should be updated with the key and IV thoroughly and this process requires a lot of computations. So it is undesirable to use HC-256 in the applications where key (or IV) is updated frequently.

8.5 Conclusion

In this chapter, we proposed a software-efficient stream cipher HC-256. Our analysis shows that HC-256 is very secure. However, the extensive security analysis of any new cipher requires a lot of efforts from many researchers. We thus invite and encourage the readers to analyze the security of HC-256.

Finally we explicitly state that HC-256 is available royalty-free and HC-256 is not covered by any patent in the world.

Chapter 9

The Stream Cipher HC-128

Abstract. HC-128 is a software-efficient stream cipher with 128-bit key and 128-bit initialization vector. HC-128 is the reduced version of HC-256. The encryption speed of the C implementation of HC-128 is about 3.05 cycles/byte on the Intel Pentium M processor.

9.1 Introduction

The stream cipher HC-128 is the simplified version of HC-256 [128] for 128-bit security. HC-128 is a simple, secure, software-efficient cipher and it is freely-available.

HC-128 consists of two secret tables, each one with 512 32-bit elements. At each step we update one element of a table with a nonlinear feedback function. All the elements of the two tables get updated every 1024 steps. At each step, one 32-bit output is generated from the non-linear output filtering function.

HC-128 is suitable for the modern (and future) superscalar microprocessors. The dependency between operations in HC-128 is very small: three consecutive steps can be computed in parallel; at each step, the feedback and output functions can be computed in parallel. The high degree of parallelism allows HC-128 to run efficiently on modern processors. We implemented HC-128 in C, and the encryption speed of HC-128 reaches 3.05 cycles/byte on the Intel Pentium M processor.

HC-128 is very secure. Our analysis shows that recovering the key of HC-128 is as difficult as exhaustive key search. To distinguish the keystream from random, we expect that more than 2^{64} keystream bits are required (our current analysis shows that about 2^{151} outputs are needed in the distinguishing attack).

This chapter is organized as follows. We introduce HC-128 in Sect. 9.2. The

security analysis of HC-128 is given in Sect. 9.3 and Sect. 9.4. Section 9.5 discusses the implementation and performance of HC-128. Section 9.6 concludes this chapter.

9.2 Cipher Specifications

In this section, we describe the stream cipher HC-128. From a 128-bit key and a 128-bit initialization vector, it generates a keystream with length up to 2^{64} bits.

9.2.1 Operations, variables and functions

Most of the operations used in HC-128 are identical to that used in HC-256 (Chapter 8), except that \boxminus is redefined as

 \exists : $x \boxminus y$ means $x - y \mod 512$

Two tables P and Q are used in HC-128. The key and the initialization vector of HC-128 are denoted as K and IV. We denote the keystream being generated as s, the same as that given in Chapter 8.

| P | : | a table with 512 32-bit elements. Each element is denoted as $P[i]$ |
|----|---|---|
| | | with $0 \le i \le 511$. |
| Q | : | a table with 512 32-bit elements. Each element is denoted as $Q[i]$ |
| | | with $0 \le i \le 511$. |
| K | : | the 128-bit key of HC-128. |
| IV | : | the 128-bit initialization vector of HC-128. |

There are six functions being used in HC-128. $f_1(x)$ and $f_2(x)$ are the same as the $\sigma_0^{\{256\}}(x)$ and $\sigma_1^{\{256\}}(x)$ being used in the message schedule of SHA-256 [100]. For $h_1(x)$, the table Q is used as S-box. For $h_2(x)$, the table P is used as S-box.

$$\begin{array}{rcl} f_1(x) &=& (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3) \\ f_2(x) &=& (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10) \\ g_1(x,y,z) &=& ((x \ggg 10) \oplus (z \ggg 23)) + (y \ggg 8) \\ g_2(x,y,z) &=& ((x \lll 10) \oplus (z \lll 23)) + (y \lll 8) \\ h_1(x) &=& Q[x_0] + Q[256 + x_2] \\ h_2(x) &=& P[x_0] + P[256 + x_2] \end{array}$$

where $x = x_3 \parallel x_2 \parallel x_1 \parallel x_0$, x is a 32-bit word, x_0 , x_1 , x_2 and x_3 are four bytes. x_3 and x_0 denote the most significant byte and the least significant byte of x, respectively.

9.2.2 Initialization process (key and IV setup)

The initialization process of HC-128 consists of expanding the key and initialization vector into P and Q (similar to the message setup in SHA-256) and running the cipher 1024 steps (with the outputs being used to update P and Q).

1. Let $K = K_0 \parallel K_1 \parallel K_2 \parallel K_3$ and $IV = IV_0 \parallel IV_1 \parallel IV_2 \parallel IV_3$, where each K_i and IV_i denotes a 32-bit number. Let $K_{i+4} = K_i$, and $IV_{i+4} = IV_i$ for $0 \le i < 4$. The key and IV are expanded into an array W_i ($0 \le i \le 1279$) as:

$$W_{i} = \begin{cases} K_{i} & 0 \le i \le 7\\ IV_{i-8} & 8 \le i \le 15\\ f_{2}(W_{i-2}) + W_{i-7} + f_{1}(W_{i-15}) + W_{i-16} + i & 16 \le i \le 1279 \end{cases}$$

2. Update the tables P and Q with the array W.

| P[i] | = | W_{i+256} | for $0 \le i \le 511$ |
|------|---|-------------|-----------------------|
| Q[i] | = | W_{i+768} | for $0 \le i \le 511$ |

3. Run the cipher 1024 steps and use the outputs to replace the table elements as follows ("⊟" denotes "−" modulo 512).

for i = 0 to 511, do $P[i] = (P[i] + g_1(P[i \boxminus 3], P[i \boxminus 10], P[i \boxminus 511])) \oplus h_1(P[i \boxminus 12]);$ for i = 0 to 511, do $Q[i] = (Q[i] + g_2(Q[i \boxminus 3], Q[i \boxminus 10], Q[i \boxminus 511])) \oplus h_2(Q[i \boxminus 12]);$

The initialization process completes and the cipher is ready to generate keystream.

9.2.3 The keystream generation algorithm

At each step, one element of a table is updated and one 32-bit output is generated. Each S-box is used to generate only 512 outputs, then it is updated in the next 512 steps. The keystream generation algorithm of HC-128 is given below (" \square " denotes "–" modulo 512, s_i denotes the output of the *i*-th step).

```
i = 0;
repeat until enough keystream bits are generated.
{
      j = i \mod{512};
      if (i \mod 1024) < 512
      {
            P[j] = P[j] + g_1(P[j \boxminus 3], P[j \boxminus 10], P[j \boxminus 511]);
            s_i = h_1(P[j \boxminus 12]) \oplus P[j];
      }
      else
      {
            Q[j] = Q[j] + g_2(Q[j \boxminus 3], Q[j \boxminus 10], Q[j \boxminus 511]);
            s_i = h_2(Q[j \boxminus 12]) \oplus Q[j];
      }
      end-if
      i = i + 1;
}
end-repeat
```

9.2.4 Encryption and decryption

The keystream is XORed with the message for encryption. The decryption is to XOR the keystream with the ciphertext. The test vectors of HC-128 are given in Appendix B.2.

9.3 Security Analysis of HC-128

The security analysis of HC-128 is similar to that of HC-256. The output and feedback functions of HC-128 are non-linear, so it is impossible to apply the fast correlation attacks [89, 57, 93, 33, 75] and algebraic attacks [3, 38, 40, 39] to recover the secret key of HC-128. The large secret S-box of HC-128 is updated during the keystream generation process, so it is very difficult to develop linear relations linking the input and output bits of the S-box.

In this section, we will analyze the period of HC-128, the security of the secret key and the security of the initialization process. The randomness of the keystream will be analyzed separately in Sect. 9.4.

9.3.1 Period

The 32778-bit state of HC-128 ensures that the period of the keystream is extremely large. But the exact period of HC-128 is difficult to predict. The average period of the keystream is estimated to be much more than 2^{256} . The large number of states also eliminates the threat of the time-memory-data tradeoff attack on stream ciphers [21] (also [7, 59]).

9.3.2 Security of the secret key

We note that the output function and the feedback function of HC-128 are nonlinear. The non-linear output function leaks a small amount of partial information at each step. The non-linear feedback function ensures that the secret key can not be recovered from those leaked partial information.

9.3.3 Security of the initialization process (key/IV setup)

The initialization process of the HC-128 consists of two stages, as given in Sect. 9.2.2. We expand the key and IV into P and Q. At this stage, every bit of the key/IV affects all the bits of the two tables and any difference in the related keys/IVs results in uncontrollable differences in P and Q. Note that the constants in the expansion function at this stage play a significant role in reducing the effect of related keys/IVs. After the expansion, we run the cipher 1024 steps and use the outputs to update the P and Q. After the initialization process, we expect that any difference in the keys/IVs would not result in biased keystream.

9.4 Randomness of the keystream

Our initial analysis shows that the distinguishing attack on HC-128 requires more than 2^{128} outputs. The analysis is given below.

We recall that at the *i*th step, if $(i \mod 1024) < 512$, the table *P* is updated as

 $P[i \mod 512] = P[i \mod 512] + g_1(P[i \boxminus 3], P[i \boxminus 10], P[i \boxminus 511]).$

We know that $s_i = h_1(P[i \boxminus 12]) \oplus P[i \mod 512]$. For $10 \le (i \mod 1024) < 511$, this feedback function can be written alternatively as

$$s_{i} \oplus h_{1}(z_{i}) = (s_{i-1024} \oplus h'_{1}(z_{i-1024})) + g_{1}(s_{i-3} \oplus h_{1}(z_{i-3}), s_{i-10} \oplus h_{1}(z_{i-10}), s_{i-1023} \oplus h'_{1}(z_{i-1023})),$$
(9.1)

where $h_1(x)$ and $h'_1(x)$ indicate two different functions since they are related to different S-boxes; z_j denotes the $P[j \boxminus 12]$ at the *j*-th step.

We note that there are two '+' operations in the feedback function. We will first investigate the least significant bits (after rotations) in the feedback function

since they are not affected by the '+' operations. Denote the *i*-th least significant bit of a as a^i . From (9.1), we obtain that for $10 \le (i \mod 1024) < 511$,

$$s_{i}^{0} \oplus s_{i-1024}^{0} \oplus s_{i-3}^{10} \oplus s_{i-10}^{8} \oplus s_{i-1023}^{23}$$

= $(h_{1}(z_{i}))^{0} \oplus (h'_{1}(z_{i-1024}))^{0} \oplus (h_{1}(z_{i-3}))^{10} \oplus$
 $\oplus (h_{1}(z_{i-10}))^{8} \oplus (h'_{1}(z_{i-1023}))^{23}.$ (9.2)

Similarly, for $1024 \times \alpha + 10 \leq i, j < 1024 \times \alpha + 511$ and $j \neq i$, we obtain

$$s_{j}^{0} \oplus s_{j-1024}^{0} \oplus s_{j-3}^{10} \oplus s_{j-10}^{8} \oplus s_{j-1023}^{23}$$

= $(h_{1}(z_{j}))^{0} \oplus (h'_{1}(z_{j-1024}))^{0} \oplus (h_{1}(z_{j-3}))^{10} \oplus$
 $\oplus (h_{1}(z_{j-10}))^{8} \oplus (h'_{1}(z_{j-1023}))^{23}.$ (9.3)

For the left sides of (9.2) and (9.3) to be equal, i.e., for the following equation

$$s_{i}^{0} \oplus s_{i-1024}^{0} \oplus s_{i-3}^{10} \oplus s_{i-10}^{23} \oplus s_{i-1023}^{23} = s_{j}^{0} \oplus s_{j-1024}^{0} \oplus s_{j-3}^{10} \oplus s_{j-10}^{23} \oplus s_{j-1023}^{23}$$
(9.4)

to hold, we require that

$$(h_{1}(z_{i}))^{0} \oplus (h'_{1}(z_{i-1024}))^{0} \oplus (h_{1}(z_{i-3}))^{10} \\ \oplus (h_{1}(z_{i-10}))^{8} \oplus (h'_{1}(z_{i-1023}))^{23} \\ = (h_{1}(z_{j}))^{0} \oplus (h'_{1}(z_{j-1024}))^{0} \oplus (h_{1}(z_{j-3}))^{10} \\ \oplus (h_{1}(z_{j-10}))^{8} \oplus (h'_{1}(z_{j-1023}))^{23}.$$

$$(9.5)$$

Approximate (9.5) as

$$H(x_1) = H(x_2),$$
 (9.6)

where H denotes a random secret 80-bit-to-1-bit S-box, x_1 and x_2 are two 80-bit random inputs, $x_1 = \overline{z}_i \| \overline{z}_{i-3} \| \overline{z}_{i-10} \| \overline{z}_{i-1023} \| \overline{z}_{i-1024}$ and $x_2 = \overline{z}_j \| \overline{z}_{j-3} \| \overline{z}_{j-10} \| \overline{z}_{j-1023} \| \overline{z}_{j-1024}$, where \overline{z} indicates the concatenation of the least significant byte and the second most significant byte of z. According to Theorem 8.3.1, (9.6) holds with probability $\frac{1}{2} + 2^{-81}$. So (9.4) holds with probability $\frac{1}{2} + 2^{-81}$. After testing the validity of 2^{164} equations (9.4), the output of the cipher can be distinguished from random with success rate 0.9772 (with false negative rate and false positive rate 0.0228). Note that only about 2^{17} equations (9.4) can be obtained from every 512 outputs, so this distinguishing attack requires about 2^{156} outputs.

We note that the attack above only deals with the least significant bit in (9.1). It may be possible to consider the rest of the 31 bits bit-by-bit. But due to the

effect of the two '+' operations in the feedback function, the attack exploiting those 31 bits is not as effective as that exploiting the least significant bit. Thus more than 2^{151} outputs are needed in this distinguishing attack.

It may be possible that the distinguishing attack against HC-128 can be improved in the future. However, it is very unlikely that our security goal can be breached since the security margin is extremely large. We thus conjecture that it is computationally infeasible to distinguish 2^{64} bits keystream of HC-128 from random.

9.5 Implementation and Performance of HC-128

The optimized implementation of HC-128 is similar to that of HC-256. On the Pentium M processor, the speed of HC-128 reaches 3.05 cycles/bye, while the speed of HC-256 is about 4.4 cycles/byte.

9.5.1 The optimized implementation of HC-128

In the optimized code, loop unrolling is used and only one branch decision is made for every 16 steps. The details of the implementation are given below. The feedback function of P is given as

 $P[i \mod 512] = P[i \mod 512] + P[i \boxminus 10] + g_1(P[i \boxminus 3], P[i \boxminus 511]).$

A register X containing 16 elements is introduced for P. If $(i \mod 1024) < 512$ and $i \mod 16 = 0$, then at the begining of the *i*th step, $X[j] = P[(i - 16 + j) \mod 512]$ for $j = 0, 1, \dots 15$, i.e. the X contains the values of $P[i \boxminus 16], P[i \boxminus 15], \dots, P[i \boxminus 11]$. In the 16 steps starting from the *i*th step, the P and X are updated as

$$\begin{split} P[i] &= P[i] + g_1(X[13], X[6], P[i+1]); \\ X[0] &= P[i]; \\ P[i+1] &= P[i+1] + g_1(X[14], X[7], P[i+2]); \\ X[1] &= P[i+1]; \\ P[i+2] &= P[i+2] + g_1(X[15], X[8], P[i+3]); \\ X[2] &= P[i+2]; \\ P[i+3] &= P[i+3] + g_1(X[0], X[9], P[i+4]); \\ X[3] &= P[i+3]; \\ & \dots \\ P[i+14] &= P[i+14] + g_1(X[11], X[4], P[i+15]); \\ X[14] &= P[i+14]; \end{split}$$

 $P[i+15] = P[i+15] + g_1(X[12], X[5], P[(i+1) \mod 512]);$ X[15] = P[i+15];

Note that at the *i*th step, two elements of $P[i \boxminus 10]$ and $P[i \boxminus 3]$ can be obtained directly from X. Also for the output function $s_i = h_1(P[i \boxminus 12]) \oplus P[i \mod 1024]$, the $P[i \boxminus 12]$ can be obtained from X. In this implementation, there is no need to compute $i \boxminus 3$, $i \boxminus 10$ and $i \boxminus 12$.

A register Y with 16 elements is used in the implementation of the feedback function of Q in the same way as that given above.

9.5.2 The performance of HC-128

Encryption Speed. We use the C codes submitted to the eStream to measure the encryption speed. The processor used in the measurement is the Intel Pentium M (1.6 GHz, 32 KB Level 1 cache, 2 MB Level 2 cache).

Using the eStream performance testing framework, the highest encryption speed of HC-128 is 3.05 cycles/byte with the compiler gcc (there are three optimization options leading to this encryption speed: k8 O3-ual-ofp, prescott O2-ofp and athon O3-ofp). Using the Intel C++ Compiler 9.1 in Windows XP (SP2), the speed is 3.3 cycles/byte. Using the Microsoft Visual C++ 6.0 in Windows XP (SP2), the speed is 3.6 cycles/byte.

Initialization Process. The key setup of HC-128 requires about 27,300 clock cycles. There are two large S-boxes in HC-128. In order to eliminate the threat of a related key/IV attack, the tables should be updated with the key and IV thoroughly and this process requires a lot of computations. It is thus undesirable to use HC-128 in the applications where key (or IV) is updated very frequently.

9.6 Conclusion

In this chapter, a software-efficient stream cipher HC-128 is described. Our analysis shows that HC-128 is very secure. However, the extensive security analysis of any new cipher requires a lot of efforts from many researchers. We encourage the readers to analyze the security of HC-128. HC-128 is not covered by any patent and it is freely available.

Chapter 10

Conclusions

In this thesis, we have reviewed the design and analysis of stream ciphers, given our own view on various techniques, then presented attacks on seven eSTREAM candidates and the design of HC-256 and HC-128.

Our attacks recover the key of seven eSTREAM stream ciphers much faster than brute force attacks. We studied the properties of additions and successfully break Phelix and ABC v2. We applied differential cryptanalysis, linear cryptanalysis and side channel attack to break the resynchronization part of DECIM, LEX, Py, Pypy and WG. Most of the attacks exploit the insufficient/improper diffusion and confusion in those ciphers. The attacks show that good diffusion and confusion are very important for stream ciphers. The use of confusion and diffusion in block ciphers is almost perfectly understood after differential and linear cryptanalysis were developed. However, very little systematic research has been published on the use of confusion and diffusion in stream ciphers in the literature. In general, it would be much harder to study the use of confusion and diffusion in stream ciphers than that in block ciphers due to the output function in stream ciphers. We believe that this topic is interesting and important. The use of confusion and diffusion in resynchronization in stream ciphers is similar to the use of confusion and diffusion in block ciphers, but there is a subtle difference. The research on this problem is important for the design of secure and efficient resynchronization schemes, and important for the design of secure and efficient self-synchronizing stream ciphers. The use of confusion and diffusion in keystream generation is expected to be even harder to analyze. The research on this problem is closely related to the design of efficient and secure stream ciphers.

We proposed HC-256 and HC-128 that are based on large, secret and changing lookup tables. Both ciphers are very fast and strong. We expect that using large, secret and changing lookup tables in stream cipher design is an efficient

approach to resist all the known and unknown attacks. Similarly using large and secret lookup tables in block cipher design is an efficient approach to resist all the attacks on block ciphers, such as the Blowfish design. Such design approaches can significantly reduce the cipher design and evaluation cost. We are in the era that an extremely strong symmetric key cipher can be designed easily.

For ciphers with large lookup tables, the cache-timing attack is a potential side-channel threat to their software implementations. However, a stream cipher with nonlinear changing secret state is quite strong against side-channel attacks because it is extremely difficult to combine the side-channel information to attack the changing state. To apply cache-timing attack to the keystream generation of a stream cipher with changing secret state, the adversary must be able to enforce the same IV to be used for many times for the same key, and must be able to change the content of the cache in order to detect the cache miss (or cache hit) of the stream cipher process running in the CPU. Thus it is highly impractical to perform a cache-timing attack against the keystream generation of a stream cipher with changing secret state, such as HC-256 and HC-128. Furthermore, the resynchronization of HC-256 and HC-128 is immune to the cache-timing attack since lookup tables are not used during the first stage of their key/IV setups.

Using large, secret and changing lookup tables is not suitable for a constrained hardware environment. The challenge is to design extremely secure and hardware efficient stream ciphers. Since the number of operations is small in the hardware efficient ciphers, it requires a lot of analysis to ensure that the cipher is secure. We expect that a substantial amount of research on this challenge is still needed.

No stream cipher with authentication was selected in the eSTREAM project. A stream cipher with authentication is useful for hardware constrained environment. The reuse of IV in such a stream cipher leaks small amount of information of the message, and it would give the false impression that the IV could be reused. But the reuse of IV is dangerous to the key, as demonstrated in our attack against Phelix. We would recommend using a stream cipher together with a MAC, and using different keys in the stream cipher and the MAC so that their security would not affect each another. It is important to design extremely efficient MACs.

Bibliography

- F. Armknecht. "Improving Fast Algebraic Attacks." In Fast Software Encryption – FSE 2004, LNCS 3017, pp. 65-82, Springer-Verlag, 2004.
- [2] F. Armknecht, C. Carlet, P. Gaborit, S. Kunzli, W. Meier and O. Ruatta. "Efficient Computation of Algebraic Immunity for Algebraic and Fast Algebraic Attacks." In Advances in Cryptology – Eurocrypt 2006, LNCS 4004, pp. 147-164, Springer-Verlag, 2006.
- [3] F. Armknecht and M. Krause. "Algebraic Attacks on Combiners with Memory." In Advances in Cryptology – Crypto 2003, LNCS 2729, pp. 162-75, Springer-Verlag, 2003.
- [4] V. Anashin, A. Bogdanov and I. Kizhvatov. "ABC: A New Fast Flexible Stream Cipher." Available at http://www.ecrypt.eu.org/stream/ciphers/abc/ abc.pdf.
- [5] V. Anashin, A. Bogdanov and I. Kizhvatov. "Security and Implementation Properties of ABC v2." In SASC 2006 - Stream Ciphers Revisited, pp. 278-292, 2006. Available at http://www.ecrypt.eu.org/stream/papersdir/2006/026.pdf.
- [6] F. Armknecht, J. Lano and B. Preneel. "Extending the Resynchronization Attack." In *Selected Areas in Cryptography – SAC 2004*, LNCS 3357, Springer-Verlag, pp. 19-38, 2004.
- [7] S. Babbage. "A Space/Time Tradeoff in Exhaustive Search Attacks on Stream Ciphers." European Convention on Security and Detection, IEE Conference publication, No. 408, May 1995.
- [8] S. Babbage and M. Dodd. "The MICKEY Stream Ciphers." In New Stream Cipher Designs - The eSTREAM Finalists, LNCS 4986, pp. 191-209, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/mickeyp3.html
- [9] E. Barkan, E. Biham and N. Keller. "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication." In Advances in Cryptology – CRYPTO 2003, LNCS 2729, pp. 600-616, Springer-Verlag, 2003.
- [10] M. Bellare, R. Canetti and H. Krawczyk. "Keying Hash Functions for Message Authentication." In Advances in Cryptology – Crypto 96, LNCS 1109, pp. 1-15, Springer-Verlag, 1996. See also RFC 2104, "HMAC: Keyed-Hashing for Message Authentication", February 1997.

- [11] C. Berbain, O. Billet, A. Canteaut, N. Courtois, B. Debraize, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin and H. Sibert. "Decim A New Stream Cipher for Hardware Applications." *ECRYPT Stream Cipher Project Report 2005/004.* Available at http://www.ecrypt.eu.org/stream/
- [12] C. Berbain, O. Billet, A. Canteaut, N. Courtois, B. Debraize, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin and H. Sibert. "DECIM v2." ECRYPT Stream Cipher Project Report 2006/004. Available at http://www.ecrypt.eu.org/stream/
- [13] C. Berbain, O. Billet, A. Canteaut, N. Courtois, B. Debraize, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin and H. Sibert. "Sosemanuk, a Fast Software-Oriented Stream Cipher." In New Stream Cipher Designs The eSTREAM Finalists, LNCS 4986, pp. 98-118, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/sosemanukp3.html
- [14] C. Berbain and H. Gilbert. "Cryptanalysis of ABC." Available at http://www. ecrypt.eu.org/stream/papersdir/048.pdf
- [15] C. Berbain, H. Gilbert and A. Maximov. "Cryptanalysis of Grain." In Fast Software Encryption – FSE 2006, LNCS 4047, pp. 15-29, Springer-Verlag, 2006.
- [16] C. Berbain, H. Gilbert and J. Patarin. "QUAD: a Practical Stream Cipher with Provable Security." In Advances in Cryptology – Eurocrypt 2006, LNCS 4004, pp. 109-128, 2006.
- [17] T. Berger, F. Arnault and C. Lauradoux. "F-FCSR Stream Ciphers." In New Stream Cipher Designs - The eSTREAM Finalists, LNCS 4986, pp. 170-178, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/ ffcsrp3.html
- [18] E. R. Berlekamp. Chapter 7 in Algorithmic Coding Theory. New York: McGraw-Hill, 1968.
- [19] D. J. Bernstein. "The Salsa 20 Family of Stream Ciphers." In New Stream Cipher Designs – The eSTREAM Finalists, LNCS 4986, pp. 84-97, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/salsa20p3.html
- [20] A. Biryukov. "A New 128-bit Key Stream Cipher LEX." In ECRYPT Stream Cipher Project Report 2005/013. Available at http://www.ecrypt.eu.org/stream/
- [21] A. Biryukov and A. Shamir. "Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers." In Advances in Cryptology – Asiacrypt 2000, LNCS 1976, pp. 1-13, Springer-Verlag, 2000.
- [22] A. Biryukov, A. Shamir and D. Wagner. "Real Time Cryptanalysis of A5/1 on a PC." In Fast Software Encryption – FSE 2000, LNCS 1978, pp. 1-18, Springer-Verlag, 2000.
- [23] A. Biryukov and D. Wagner. "Slide Attacks." In Fast Software Encryption FSE'99, LNCS 1636, pp. 245-259, Springer-Verlag, 1999.

- [24] T. Beth and F.C. Piper. "The Stop-And-Go-Generator." In Advances in Cryptology – Eurocrypt'84, LNCS 209, pp. 88-92, Springer-Verlag, 1984.
- [25] E. Biham and J. Seberry. "Py (Roo): A Fast and Secure Stream Cipher Using Rolling Arrays." The ECRYPT eSTREAM project Phase 2 focus ciphers. Available at http://www.ecrypt.eu.org/stream/ciphers/py/py.ps.
- [26] E. Biham and J. Seberry. "Pypy (Roopy): Another Version of Py." The ECRYPT eSTREAM project Phase 2 focus ciphers. Available at http://www.ecrypt.eu.org/ stream/p2ciphers/py/pypy_p2.ps
- [27] E. Biham and A. Shamir. "Differential Cryptanalysis of DES-like Cryptosystems." In Advances in Cryptology – Crypto'90, LNCS 537, pp. 2-21, Springer-Verlag, 1991.
- [28] S. R. Blackburn. "Fast Rational Interpolation, Reed-Solomon Decoding and the Linear Complexity Profiles of Sequences." *IEEE Transactions on Information Theory*, vol. 43, no. 2, pp.537-548, March 1997.
- [29] R. E. Blahut. Theory and Practice of Error Control Codes, Addison-Wesley, 1983.
- [30] M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen and O. Scavenius.
 "Rabbit: A New High-Performance Stream Cipher." In *Fast Software Encryption* - FSE 2003, LNCS 2887, pp. 307-329, Springer-Verlag, 2003.
- [31] Y. Borissov, S. Nikova, B. Preneel and J. Vandewalle. "On a Resynchronization Weakness in a Class of Combiners with Memory." In *Security in Communication Networks – SCN 2002*, LNCS 2576, pp. 164-173, Springer-Verlag, 2002.
- [32] A. Canteaut and M. Trabbia. "Improved Fast Correlation Attacks Using Parity-Check Equations of Weight 4 and 5." In Advances in Cryptology – EUROCRYPT 2000, LNCS 1807, pp. 573-588, Springer-Verlag, 2000.
- [33] V.V. Chepyzhov, T. Johansson and B. Smeets. "A Simple Algorithm for Fast Correlation Attacks on Stream Ciphers." In *Fast Software Encryption – FSE* 2000, LNCS 1978, pp. 181-195, Springer-Verlag, 2000.
- [34] D. Coppersmith, H. Krawczyk and Y. Mansour. "The Shrinking Generator." In Advances in Cryptology – CRYPTO'93, LNCS 773, pp. 22-39. Springer-Verlag, 1993.
- [35] D. Coppersmith, S. Halevi and C. Jutla. "Scream: A Software-Efficient Stream Cipher." In Fast Software Encryption – FSE 2002, LNCS 2365, pp. 195-209, Springer-Verlag, 2002.
- [36] D. Coppersmith, S. Halevi and C. Jutla. "Cryptanalysis of Stream Ciphers with Linear Masking." In Advances in Cryptology – Crypto 2002, LNCS 2442, pp. 515-532, Springer-Verlag, 2002.
- [37] P. Crowley. "Improved Cryptanalysis of Py." Available at http://www.ecrypt. eu.org/stream/papersdir/2006/010.pdf
- [38] N. Courtois. "Higher Order Correlation Attacks, XL algorithm and Cryptanalysis of Toyocrypt." In *Information Security and Cryptology – ICISC 2002*, LNCS 2587, pp. 182-199, Springer-Verlag, 2002.

- [39] N. T. Courtois. "Fast Algebraic Attacks on Stream Ciphers with Linear Feedback." In Advances in Cryptology – CRYPTO 2003, LNCS 2729, pp. 176-194, Springe-Verlag, 2003.
- [40] N. T. Courtois and W. Meier. "Algebraic Attacks on Stream Ciphers with Linear Feedback." In Advances in Cryptology – EUROCRYPT 2003, LNCS 2656, pp. 345-359, Springer-Verlag, 2003.
- [41] J. Daemen, R. Govaerts and J. Vandewalle. "Resynchronization Weakness in Synchronous Stream Ciphers." In Advances in Cryptology – EUROCRYPT'93, LNCS 765, pp. 159-167, Springer-Verlag, 1994.
- [42] J. Daemen, L. Knudsen and V. Rijmen. "The Block Cipher Square." In Fast Software Encryption – FSE'97, LNCS 1267, pp. 149165, 1997.
- [43] J. Daeman and V. Rijmen. "AES Proposal: Rijndael." Available on-line from NIST at http://csrc.nist.gov/encryption/aes/rijndael/
- [44] C. De Cannière and B. Preneel. "Trivium." In New Stream Cipher Designs -The eSTREAM Finalists, LNCS 4986, pp. 244-266, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/triviump3.html
- [45] ECRYPT Stream Cipher Project, at http://www.ecrypt.eu.org/stream/
- [46] ECRYPT Stream Cipher Project, Call for Stream Cipher Primitives, http://www.ecrypt.eu.org/stream/call/
- [47] P. Ekdahl and T. Johansson. "A New Version of the Stream Cipher SNOW." In Selected Areas in Cryptology – SAC 2002, LNCS 2595, pp. 47-61, Springer-Verlag, 2002.
- [48] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks and T. Kohno. "Helix, Fast Encryption and Authentication in a Single Cryptographic Primitive." In *Fast Software Encryption – FSE 2003*, LNCS 2887, pp. 330-346, Springer-Verlag, 2003.
- [49] P. Flajolet and A.M. Odlyzko. "Random Mapping Statistics." In Advances in Cryptology – Eurocrypt'89, LNCS 434, pp. 329-354, Springer-Verlag, 1990.
- [50] S. Fluhrer, I. Mantin and A. Shamir. "Weaknesses in the Key Scheduling Algorithm of RC4." In *Selected Areas in Cryptography – SAC 2001*, LNCS 2259, pp. 1-24, Springer-Verlag, 2001.
- [51] S. Fluhrer and D. McGrew. "Statistical Analysis of the Alleged RC4 Keystream Generator." In *Fast Software Encryption – FSE 2000*, LNCS 1978, pp. 19-30, 2001.
- [52] W. Freeman, G. Sullivan and F. Weierud. "Purple Revealed: Simulation and Computer-Aided Cryptanalysis of Angooki Taipu B." *Cryptologia* 27(1), January 2003. pp. 143.
- [53] K. Gaj and A. Orlowski. "Facts and Myths of Enigma: Breaking Stereotypes." Advances in Cryptology – Eurocrypt 2003, LNCS 2656, pp. 106-122, Springer-Verlag, 2003.

- [54] B.M. Gammel, R. Göttfert and O. Kniffler. "The Achterbahn stream cipher." eSTREAM, ECRYPT Stream Cipher Project. Available at http://www.ecrypt.eu. org/stream/achterbahn.html
- [55] B.M. Gammel, R. Göttfert and O. Kniffler. "Achterbahn-128/80." eSTREAM, ECRYPT Stream Cipher Project. Available at http://www.ecrypt.eu.org/ stream/achterbahnp2.html
- [56] D. Gligoroski, S. Markovski, L. Kocarev and M. Gusev. "Stream Cipher Edon80." Available at http://www.ecrypt.eu.org/stream/edon80p3.html
- [57] J. D. Golić. "Towards Fast Correlation Attacks on Irregularly Clocked Shift Registers." In Advances in Cryptography – Eurocrypt'95, pp. 248-262, Springer-Verlag, 1995.
- [58] J. D. Golić. "Linear Models for Keystream Generator." IEEE Trans. on Computers, 45(1):41-49, Jan 1996.
- [59] J. D. Golić. "Cryptanalysis of Alleged A5 Stream Cipher." In Advances in Cryptology – Eurocrypt'97, LNCS 1233, pp. 239-255, Springer-Verlag, 1997.
- [60] J. D. Golić. "Linear Statistical Weakness of Alleged RC4 Keystream Generator." In Advancesin Cryptology – Eurocrypt'97, pp. 226-238, Springer-Verlag, 1997.
- [61] J. D. Golić, G. Morgari. "On the Resynchronization Attack." In Fast Software Encryption – FSE 2003, LNCS 2887, pp. 100-110, Springer-Verlag, 2003.
- [62] G. Gong and A. Youssef. "Cryptographic Properties of the Welch-Gong Transformation Sequence Generators." *IEEE Transactions on Information Theory*, vol. 48, No. 11, pp. 2837-2846, Nov. 2002.
- [63] L. K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search." Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, pp. 212-219, 1996.
- [64] C. G. Günther. "Alternating Step Generators Controlled by de Bruijn Sequences." In Advances in Cryptology – EuroCrypt'87, LNCS 304, pp. 5-14, Springer-Verlag, 1987.
- [65] P. Hawkes and G. G. Rose. "Rewriting Variables: The Complexity of Fast Algebraic Attacks on Stream Ciphers." In Advances in Cryptology – CRYPTO 2004, LNCS 3152, pp. 390-406, Springer-Verlag, 2004.
- [66] M. Hell, T. Johansson and W. Meier. "Grain A Stream Cipher for Constrained Environments." eSTREAM, ECRYPT Stream Cipher Project. Available at http: //www.ecrypt.eu.org/stream/grain.html
- [67] M. Hell, T. Johansson, A. Maximov and W. Meier. "The Grain Family of Stream Ciphers." In New Stream Cipher Designs – The eSTREAM Finalists, LNCS 4986, pp. 179-190, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu. org/stream/grainp3.html
- [68] M. Hell and T. Johansson. "Cryptanalysis of Achterbahn-Version 2." Selected Areas in Cryptography – SAC 2006, LNCS 4356, pp. 45-55, Springer-Verlag, 2007.

- [69] M. Hell and T. Johansson. "Cryptanalysis of Achterbahn-128/80." Available at http://www.ecrypt.eu.org/stream/papersdir/2006/054.pdf
- [70] M. Hell and T. Johansson. "A Key Recovery Attack on Edon80." In Advances in Cryptology – ASIACRYPT 2007, LNCS 4833, pp. 568-581, Springer-Verlag, 2007.
- [71] M. E. Hellman. "A Cryptanalytic Time-Memory Trade-Off." *IEEE Transactions on Information Theory*, vol. IT-26, No. 4, pp. 401-406, July 1980.
- [72] R. J. Jenkins Jr., "ISAAC." In Fast Software Encryption FSE'96, LNCS 1039, pp. 41-49, Springer-Verlag, 1996.
- [73] T. Johansson and F. Jönsson. "Improved Fast Correlation Attacks on Stream Ciphers via Convolutional Codes." In Advances in Cryptology – EUROCRYPT'99, LNCS 1592, pp. 347-362, Springer-Verlag, 1999.
- [74] T. Johansson and F. Jönsson. "Fast Correlation Attacks Based on Turbo Code Techniques." In Advances in Cryptology – CRYPTO'99, LNCS 1666, pp. 181-197, Springer-Verlag, 1999.
- [75] T. Johansson and F. Jönsson. "Fast Correlation Attacks through Reconstruction of Linear Polynomials." In Advances in Cryptology – CRYPTO 2000, LNCS 1880, pp. 300-315, Springer-Verlag, 2000.
- [76] T. Johansson, W. Meier and F. Muller. "Cryptanalysis of Achterbahn." In Fast Software Encryption – FSE 2006, LNCS 4047, pp. 1-14, Springer-Verlag, 2006.
- [77] A. Joux and J. Reinhard. "Overtaking VEST." In Fast Software Encryption FSE 2007, LNCS 4593, pp. 58-72, Springer-Verlag, 2007.
- [78] N. Keller, S. D. Miller, I. Mironov and R. Venkatesan. "MV3: A New Word Based Stream Cipher Using Rapid Mixing and Revolving Buffers." In *Topics in* Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, LNCS 4377, pp. 1-19, Springer-Verlag, 2006.
- [79] A. Klimov and A. Shamir. "Cryptographic Applications of T-functions." In Selected Areas in Cryptography – SAC 2003, LNCS 3006, pp. 248-261, Springer-Verlag, 2003.
- [80] A. Klimov and A. Shamir. "New Cryptographic Primitives Based on Multiword T-functions." In *Fast Software Encryption – FSE 2004*, LNCS 3017, pp. 1-15, Springer-Verlag, 2004.
- [81] S. Khazaei. "Divide and Conquer Attack on ABC Stream Cipher." Available at http://www.ecrypt.eu.org/stream/papersdir/052.pdf
- [82] L. Knudsen, W. Meier, B. Preneel, V. Rijmen and S. Verdoolaege. "Analysis Methods for (Alleged) RC4". In Advances in Cryptology – Asiacrypt'98, LNCS 1514, pp. 327-341, Springer-Verlag, 1998.
- [83] T. Kohno. "Analysis of the WinZip Encryption Method." Available at http:// eprint.iacr.org/2004/078
- [84] I. Mantin. "A Practical Attack on the Fixed RC4 in the WEP Mode." Advances in Cryptology – ASIACRYPT 2005, LNCS 3788, pp. 395-411, Springer-Verlag, 2005.

- [85] I. Mantin. "Predicting and Distinguishing Attacks on RC4 Keystream Generator." Advances in Cryptography – EUROCRYPT 2005, LNCS 3494, pp. 491-506, Springer-Verlag, 2005.
- [86] I. Mantin and A. Shamir. "A Practical Attack on Broadcast RC4." In Fast Software Encryption – FSE 2001, LNCS 2355, pp. 152-164, Springer-Verlag, 2002.
- [87] J. L. Massey. "Shift-Register Synthesis and BCH Decoding." IEEE Transaction Information Theory, 15, 122-127, 1969.
- [88] M. Matsui. "Linear Cryptanalysis Method for DES Cipher." In Advances in Cryptology – Eurocrypt'93, LNCS 765, pp. 386-397, Springer-Verlag, 1994.
- [89] W. Meier and O. Staffelbach. "Fast Correlation Attacks on Certain Stream Ciphers." Journal of Cryptography, 1(3):159-176, 1989.
- [90] W. Meier and O. Staffelbach. "The Self-Shrinking Generator." In Advances in Cryptology – EUROCRYPT'94, LNCS 950, pp. 205-214, Springer-Verlag, 1994.
- [91] A. Menezes, P. van Oorschot and S. Vanstone. Handbook of Applied Cryptography. Available at http://www.cacr.math.uwaterloo.ca/hac/
- [92] Microsoft Office. At http://office.microsoft.com/home/default.aspx
- [93] M. Mihaljević, M.P.C. Fossorier and H. Imai. "A Low-Complexity and High-Performance Algorithm for Fast Correlation Attack." In *Fast Software Encryption* - *FSE 2000*, pp. 196-212, Springer-Verlag, 2000.
- [94] I. Mironov. "(Not So) Random Shuffles of RC4." In Advances in Cryptology Crypto 2002, LNCS 2442, pp. 304-319, Springer-Verlag, 2002.
- [95] S. Mister and S.E. Tavares. "Cryptanalysis of RC4-like Ciphers." In Selected Areas in Cryptography – SAC'98), LNCS 1556, pp. 131-143, Springer-Verlag, 1999.
- [96] F. Muller. "Differential Attacks against the Helix Stream Cipher." In Fast Software Encryption – FSE 2004, LNCS 3017, pp. 94-108, Springer-Verlag, 2004.
- [97] S. Murphy and M.J.B. Robshaw. "Key-Dependent S-boxes and Differential Cryptanalysis." Designs, Codes, and Cryptography 27(3), pp. 229-255, 2002.
- [98] National Institute of Standards and Technology. "DES Modes of Operation." Federal Information Processing Standards Publication (FIPS) 81. Available at http://csrc.nist.gov/publications/fips/
- [99] National Institute of Standards and Technology. "Data Encryption Standard." Federal Information Processing Standards Publication (FIPS) 46-3. Available at http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf
- [100] National Institute of Standards and Technology. "Secure Hash Standard (SHS)." Available at http://csrc.nist.gov/cryptval/shs.html
- [101] National Institute of Standards and Technology. "Advanced Encryption Standard (AES)." Federal Information Processing Standards Publication (FIPS) 197. Available at http://csrc.nist.gov/publications/fips/
- [102] National Security Agency. "Securing Record Communications The TSEC/KW-26." Available at http://www.nsa.gov/publications/publi00017.pdf

- [103] Y. Nawaz and G. Gong. "The WG Stream Cipher." ECRYPT Stream Cipher Project Report 2005/033. Available at http://www.ecrypt.eu.org/stream/
- [104] Y. Nawaz and G. Gong. "Preventing Chosen IV Attack on WG Cipher by Increasing the Length of Key/IV Setup." ECRYPT Stream Cipher Project Report 2005/047. Available at http://www.ecrypt.eu.org/stream/
- [105] NESSIE. "NESSIE Project Announces Final Selection of Crypto Algorithms." Available at https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/ press_release_feb27.pdf
- [106] S. Paul and B. Preneel. "A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher." In *Fast Software Encryption* - FSE 2004, LNCS 3017, pp. 245–259, Springer-Verlag, 2004.
- [107] S. Paul and B. Preneel. "Solving Systems of Differential Equations of Addition." In Australasian Conference on Information Security and Privacy – ACISP 2005, LNCS 3574, pp. 75-88, Springer-Verlag, 2005.
- [108] S. Paul and B. Preneel. "Near Optimal Algorithms for Solving Differential Equations of Addition with Batch Queries." In *Progress in Cryptology – Indocrypt* 2005, LNCS 3797, pp. 90-103, Springer-Verlag, 2003.
- [109] S. Paul, B. Preneel and S. Sekar. "Distinguishing Attack on the Stream Cipher Py." In *Fast Software Encryption – FSE 2006*, LNCS 4047, pp. 405–421, Springer-Verlag, 2006.
- [110] S. Paul and B. Preneel. "On the (In)security of Stream Ciphers Based on Arrays and Modular Addition." In Advances in Cryptology – ASIACRYPT 2006, LNCS 4284, pp. 69–83, Springer-Verlag, 2006.
- [111] M.N. Plasencia. "Cryptanalysis of Achterbahn-128/80." Available at http://www. ecrypt.eu.org/stream/papersdir/2006/055.pdf
- [112] P. Rogaway and D. Coppersmith. "A Software Optimized Algorithm." Journal of Cryptography, 11(4), pp. 273-287, 1998.
- [113] G.G. Rose and P. Hawkes. "Turing: a Fast Stream Cipher." In Fast Software Encryption – FSE'03, LNCS 2887, pp. 290-306, Springer-Verlag, 2003.
- [114] R.L. Rivest. "The RC4 Encryption Algorithm." RSA Data Security, Inc., March 12, 1992.
- [115] R. Rivest, A. Shamir and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM*, Vol. 21 (2), pp. 120-126. 1978.
- [116] B. Schneier. "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)." In Fast Software Encryption – FSE'93, LNCS 809, pp. 191-204, Springer-Verlag, 1994.
- [117] B. Schneier and D. Whiting. "Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor." In *Fast Software Encryption – FSE'97*, LNCS 1267, pp. 242-259, Springer-Verlag, 1997.

- [118] C. Shannon. "Communication Theory of Secrecy Systems." Bell System Technical Journal 28 (4): 656-715.
- [119] P. W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." In *Proceedings of the 35th Annual Sym*posium on Foundations of Computer Science, 1994.
- [120] T. Siegenthaler. "Correlation-Immunity of Nonlinear Combining Functions for Cryptographic Applications." *IEEE Transactions on Information Theory*, IT-30:776-780,1984.
- [121] T. Siegenthaler. "Decrypting a Class of Stream Ciphers Using Ciphertext Only." IEEE Transactions on Computers, C-34(1):81-85, Jan. 1985.
- [122] V. Strassen. "Gaussian Elimination is not Optimal." Numer. Math. 13, pp. 354-356, 1969.
- [123] G. Sullivan and F. Weierud. "Breaking German Army Ciphers." In *Cryptologia* 24(3), July 2005, pp. 193-232.
- [124] S. Vaudenay. "On the Weak Keys of Blowfish." Fast Software Encryption FSE'96, LNCS 1039, pp. 27-32, Springer-Verlag, 1996.
- [125] D. Watanabe, S. Furuya, H. Yoshida, K. Takaragi and B. Preneel. "A New Keystream Generator MUGI." In *Fast Software Encryption – FSE 2002*, LNCS 2365, pp. 179-194, Springer-Verlag, 2002.
- [126] D. Whiting, B. Schneier, S. Lucks and F. Muller. "Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive." eSTREAM, ECRYPT Stream Cipher Project Report 2005/027.
- [127] WinZip Computing, Inc. Homepage, Mar. 2004. Available at http://www. winzip.com/
- [128] H. Wu. "A New Stream Cipher HC-256." In Fast Software Encryption FSE 2004, LNCS 3017, pp. 226-244, Springer-Verlag, 2004. The full version is available at http://eprint.iacr.org/2004/092.pdf
- [129] H. Wu. "The Misuse of RC4 in Microsoft Word and Excel." IACR ePrint, 2005. Available at http://eprint.iacr.org/2005/007
- [130] H. Wu and B. Preneel. "Cryptanalysis of the Stream Cipher DECIM." In Fast Software Encryption – FSE 2006, LNCS 4047, pp. 30-40, Springer-Verlag, 2006.
- [131] H. Wu and B. Preneel. "Resynchronization Attacks on WG and LEX." In Fast Software Encryption – FSE 2006, LNCS 4047, pp. 422-432, Springer-Verlag, 2006.
- [132] H. Wu and B. Preneel. "Cryptanalysis of the Stream Cipher ABC v2." In Selected Areas in Cryptography – SAC 2006, LNCS 4356, pp. 56-66, Springer-Verlag, 2007.
- [133] H. Wu and B. Preneel. "Differential attacks on Stream Ciphers Phelix." In Fast Software Encryption – FSE 2007, LNCS 4593, pp. 87-100, Springer-Verlag, 2007.
- [134] H. Wu and B. Preneel. "Differential attacks on Stream Ciphers Py, Py6 and Pypy." In Advances in Cryptology – Eurocrypt 2007, LNCS 4515, pp. 276-290, Springer-Verlag, 2007.

- [135] H. Wu. "The Stream Cipher HC-128." In New Stream Cipher Designs The eSTREAM Finalists, LNCS 4986, pp. 39-47, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/hcp3.html
- [136] H. Zhang, L. Li and X. Wang. "Fast Correlation Attack on Stream Cipher ABC v3." Available at http://www.ecrypt.eu.org/stream/papersdir/2006/049.pdf
- [137] H. Zhang, S. Wang and X. Wang. "The Probability Advantages of Two Linear Expressions in Symmetric Ciphers." Available at http://www.ecrypt.eu.org/ stream/papersdir/2006/046.pdf

Appendix A

The Number of IVs to Break DECIM

Table A.1 gives the number of IVs required to break DECIM with a 64-bit IV. 44 key bits can be recovered with less than 2^{20} IVs. Table A.1 gives the number of IVs required to break DECIM with an 80-bit IV. 41 key bits can be recovered with less than 2^{21} IVs. Only the first 2 bytes of the keystream are required in the attack, and the amount of computation required in the attacks is negligible.

We explain Table A.1 with K_0 as an example. K_0 is related to s_{112} since $s_{112} = K_0 \oplus IV_0$. s_{112} is mapped to s_{192+60} with probability 0.0318 (this probability is obtained by tracing s_{112} through the initialization process). Thus K_0 could be recovered by observing the first bits of the keystreams. About $2^{18.95}$ IVs are required to achieve a success probability of 0.977.

| | Affected Bits | Number | | | Affected Bits | Number |
|---------------------|-----------------------------------|-----------|---|---------------------|-----------------------------------|-----------|
| | | of IVs | | | | of IVs |
| | | (Log_2) | | | | (Log_2) |
| K_0 | $s_{112} \Rightarrow s_{192+60}$ | 18.95 | | K_1 | $s_{57} \Rightarrow s_{192+122}$ | 20.83 |
| K_2 | $s_{58} \Rightarrow s_{192+116}$ | 18.80 | 1 | K_3 | $s_{115} \Rightarrow s_{192+104}$ | 20.46 |
| K_4 | $s_{116} \Rightarrow s_{192+105}$ | 21.41 | 1 | K_5 | $s_{117} \Rightarrow s_{192+106}$ | 21.54 |
| K_6 | $s_{118} \Rightarrow s_{192+107}$ | 21.67 | 1 | K_7 | $s_{119} \Rightarrow s_{192+108}$ | 21.72 |
| K_8 | $s_{120} \Rightarrow s_{192+145}$ | 21.21 | 1 | K_9 | $s_{121} \Rightarrow s_{192+110}$ | 21.92 |
| K_{10} | $s_{10} \Rightarrow s_{192+116}$ | 17.69 | 1 | K_{11} | $s_{11} \Rightarrow s_{192+117}$ | 19.62 |
| K_{12} | $s_{68} \Rightarrow s_{192+6}$ | 18.88 | 1 | K_{13} | $s_{69} \Rightarrow s_{192+7}$ | 20.82 |
| K_{14} | $s_{70} \Rightarrow s_{192+8}$ | 18.82 | 1 | K_{15} | $s_{127} \Rightarrow s_{192+116}$ | 16.66 |
| K_{16} | $s_{128} \Rightarrow s_{192+117}$ | 18.70 | 1 | K_{17} | $s_{17} \Rightarrow s_{192+6}$ | 16.92 |
| K_{18} | $s_{18} \Rightarrow s_{192+7}$ | 18.82 | | K_{19} | $s_{19} \Rightarrow s_{192+8}$ | 16.80 |
| K_{20} | $s_{20} \Rightarrow s_{192+9}$ | 18.73 | 1 | K_{21} | $s_{21} \Rightarrow s_{192+6}$ | 18.59 |
| K_{22} | $s_{22} \Rightarrow s_{192+7}$ | 20.67 | 1 | K_{23} | $s_{23} \Rightarrow s_{192+8}$ | 18.70 |
| K_{24} | $s_{80} \Rightarrow s_{192+146}$ | 20.80 | 1 | K_{25} | $s_{25} \Rightarrow s_{192+116}$ | 17.97 |
| K_{26} | $s_{138} \Rightarrow s_{192+6}$ | 17.79 | 1 | K_{27} | $s_{139} \Rightarrow s_{192+7}$ | 19.87 |
| K_{28} | $s_{140} \Rightarrow s_{192+8}$ | 17.86 | 1 | K_{29} | $s_{141} \Rightarrow s_{192+9}$ | 19.67 |
| K_{30} | $s_{142} \Rightarrow s_{192+10}$ | 21.46 | 1 | K_{31} | $s_{31} \Rightarrow s_{192+182}$ | 18.36 |
| K_{32} | $s_{32} \Rightarrow s_{192+183}$ | 20.70 | 1 | K_{33} | $s_{33} \Rightarrow s_{192+113}$ | 20.97 |
| K_{34} | $s_{34} \Rightarrow s_{192+114}$ | 21.03 | 1 | K_{35} | $s_{91} \Rightarrow s_{192+116}$ | 19.95 |
| K_{36} | $s_{36} \Rightarrow s_{192+116}$ | 15.55 | | K_{37} | $s_{37} \Rightarrow s_{192+117}$ | 17.56 |
| K_{38} | $s_{94} \Rightarrow s_{192+145}$ | 18.94 | | K_{39} | $s_{39} \Rightarrow s_{192+104}$ | 19.62 |
| K_{40} | $s_{152} \Rightarrow s_{192+60}$ | 16.43 | 1 | K_{41} | $s_{153} \Rightarrow s_{192+116}$ | 17.90 |
| K_{42} | $s_{154} \Rightarrow s_{192+117}$ | 19.93 | 1 | K_{43} | $s_{43} \Rightarrow s_{192+108}$ | 20.61 |
| K_{44} | $s_{156} \Rightarrow s_{192+145}$ | 16.90 |] | K_{45} | $s_{157} \Rightarrow s_{192+146}$ | 18.96 |
| K_{46} | $s_{46} \Rightarrow s_{192+35}$ | 20.45 | | K_{47} | $s_{47} \Rightarrow s_{192+6}$ | 16.68 |
| K_{48} | $s_{160} \Rightarrow s_{192+145}$ | 18.68 | 1 | K_{49} | $s_{161} \Rightarrow s_{192+181}$ | 15.59 |
| K_{50} | $s_{162} \Rightarrow s_{192+182}$ | 17.59 | | K_{51} | $s_{51} \Rightarrow s_{192+116}$ | 15.62 |
| K_{52} | $s_{52} \Rightarrow s_{192+117}$ | 17.64 | 1 | K_{53} | $s_{53} \Rightarrow s_{192+118}$ | 19.47 |
| K_{54} | $s_{54} \Rightarrow s_{192+119}$ | 20.05 | | K_{55} | $s_{55} \Rightarrow s_{192+120}$ | 20.61 |
| K_{56} | $s_{168} \Rightarrow s_{192+76}$ | 22.27 | | K_{57} | $s_{169} \Rightarrow s_{192+103}$ | 18.43 |
| K_{58} | $s_{170} \Rightarrow s_{192+104}$ | 18.17 | | K_{59} | $s_{171} \Rightarrow s_{192+105}$ | 18.93 |
| $\overline{K_{60}}$ | $s_{172} \Rightarrow s_{192+106}$ | 19.11 | | $\overline{K_{61}}$ | $s_{173} \Rightarrow s_{192+107}$ | 19.24 |
| $\overline{K_{62}}$ | $s_{174} \Rightarrow s_{192+108}$ | 19.42 | | $\overline{K_{63}}$ | $s_{175} \Rightarrow s_{192+109}$ | 19.58 |

Table A.1: Number of IVs required to recover the key bits (64-bit IV)

| | Affected Bits | Number | 1 | | Affected Bits | Number |
|-----------------|-----------------------------------|-----------|---|-----------------|-----------------------------------|-----------|
| | | of IVs | | | | of IVs |
| | | (Log_2) | | | | (Log_2) |
| K_0 | $s_{32} \Rightarrow s_{192+183}$ | 20.70 | 1 | K_1 | $s_{33} \Rightarrow s_{192+113}$ | 20.97 |
| K_2 | $s_{34} \Rightarrow s_{192+114}$ | 21.03 | 1 | K_3 | $s_{35} \Rightarrow s_{192+115}$ | 21.13 |
| K_4 | $s_{36} \Rightarrow s_{192+116}$ | 15.55 | 1 | K_5 | $s_{37} \Rightarrow s_{192+117}$ | 17.56 |
| K_6 | $s_{38} \Rightarrow s_{192+118}$ | 19.43 | 1 | K_7 | $s_{39} \Rightarrow s_{192+104}$ | 19.62 |
| K_8 | $s_{40} \Rightarrow s_{192+105}$ | 20.37 | 1 | K_9 | $s_{41} \Rightarrow s_{192+121}$ | 20.30 |
| K_{10} | $s_{42} \Rightarrow s_{192+107}$ | 20.48 | 1 | K_{11} | $s_{43} \Rightarrow s_{192+108}$ | 20.61 |
| K_{12} | $s_{44} \Rightarrow s_{192+109}$ | 20.77 | 1 | K_{13} | $s_{45} \Rightarrow s_{192+34}$ | 20.70 |
| K_{14} | $s_{46} \Rightarrow s_{192+35}$ | 20.45 | 1 | K_{15} | $s_{47} \Rightarrow s_{192+6}$ | 16.68 |
| K_{16} | $s_{48} \Rightarrow s_{192+7}$ | 18.72 | 1 | K_{17} | $s_{49} \Rightarrow s_{192+8}$ | 16.68 |
| K_{18} | $s_{50} \Rightarrow s_{192+9}$ | 18.66 | 1 | K_{19} | $s_{51} \Rightarrow s_{192+116}$ | 15.62 |
| K_{20} | $s_{52} \Rightarrow s_{192+117}$ | 17.64 | 1 | K_{21} | $s_{53} \Rightarrow s_{192+118}$ | 19.47 |
| K_{22} | $s_{54} \Rightarrow s_{192+119}$ | 20.05 | 1 | K_{23} | $s_{55} \Rightarrow s_{192+120}$ | 20.61 |
| K_{24} | $s_{56} \Rightarrow s_{192+121}$ | 20.63 |] | K_{25} | $s_{57} \Rightarrow s_{192+122}$ | 20.83 |
| K_{26} | $s_{58} \Rightarrow s_{192+116}$ | 18.80 | 1 | K_{27} | $s_{59} \Rightarrow s_{192+12}$ | 23.00 |
| K_{28} | $s_{60} \Rightarrow s_{192+13}$ | 23.41 | 1 | K_{29} | $s_{61} \Rightarrow s_{192+14}$ | 23.66 |
| K_{30} | $s_{62} \Rightarrow s_{192+15}$ | 23.78 |] | K_{31} | $s_{63} \Rightarrow s_{192+16}$ | 24.09 |
| K_{32} | $s_{64} \Rightarrow s_{192+17}$ | 24.00 | | K_{33} | $s_{65} \Rightarrow s_{192+18}$ | 24.19 |
| K_{34} | $s_{66} \Rightarrow s_{192+19}$ | 24.22 | | K_{35} | $s_{67} \Rightarrow s_{192+5}$ | 23.44 |
| K_{36} | $s_{68} \Rightarrow s_{192+6}$ | 18.88 | | K_{37} | $s_{69} \Rightarrow s_{192+7}$ | 20.82 |
| K_{38} | $s_{70} \Rightarrow s_{192+8}$ | 18.82 | | K_{39} | $s_{71} \Rightarrow s_{192+60}$ | 16.77 |
| K_{40} | $s_{72} \Rightarrow s_{192+61}$ | 18.75 | | K_{41} | $s_{73} \Rightarrow s_{192+62}$ | 20.59 |
| K_{42} | $s_{74} \Rightarrow s_{192+63}$ | 21.11 | | K_{43} | $s_{75} \Rightarrow s_{192+64}$ | 21.71 |
| K_{44} | $s_{76} \Rightarrow s_{192+65}$ | 21.67 | | K_{45} | $s_{77} \Rightarrow s_{192+66}$ | 21.85 |
| K_{46} | $s_{78} \Rightarrow s_{192+67}$ | 21.81 | | K_{47} | $s_{79} \Rightarrow s_{192+145}$ | 18.82 |
| K_{48} | $s_{80} \Rightarrow s_{192+146}$ | 20.80 | | K_{49} | $s_{81} \Rightarrow s_{192+70}$ | 22.05 |
| K_{50} | $s_{82} \Rightarrow s_{192+71}$ | 22.18 | | K_{51} | $s_{83} \Rightarrow s_{192+72}$ | 22.40 |
| K_{52} | $s_{84} \Rightarrow s_{192+73}$ | 22.43 | | K_{53} | $s_{85} \Rightarrow s_{192+74}$ | 22.42 |
| K_{54} | $s_{86} \Rightarrow s_{192+75}$ | 22.43 | | K_{55} | $s_{87} \Rightarrow s_{192+76}$ | 22.55 |
| K_{56} | $s_{88} \Rightarrow s_{192+154}$ | 24.02 | | K_{57} | $s_{89} \Rightarrow s_{192+155}$ | 24.04 |
| K_{58} | $s_{90} \Rightarrow s_{192+156}$ | 24.15 | | K_{59} | $s_{91} \Rightarrow s_{192+116}$ | 19.95 |
| K_{60} | $s_{92} \Rightarrow s_{192+117}$ | 21.97 | | K_{61} | $s_{93} \Rightarrow s_{192+118}$ | 23.77 |
| K_{62} | $s_{94} \Rightarrow s_{192+145}$ | 18.94 | | K_{63} | $s_{95} \Rightarrow s_{192+146}$ | 20.91 |
| K_{64} | $s_{96} \Rightarrow s_{192+147}$ | 22.79 | | K_{65} | $s_{97} \Rightarrow s_{192+148}$ | 23.33 |
| K_{66} | $s_{98} \Rightarrow s_{192+149}$ | 23.77 | - | K_{67} | $s_{99} \Rightarrow s_{192+150}$ | 23.64 |
| K ₆₈ | $s_{100} \Rightarrow s_{192+63}$ | 22.65 | | K_{69} | $s_{101} \Rightarrow s_{192+4}$ | 23.12 |
| K70 | $s_{102} \Rightarrow s_{192+65}$ | 23.66 | | K ₇₁ | $s_{103} \Rightarrow s_{192+178}$ | 23.80 |
| K ₇₂ | $s_{104} \Rightarrow s_{192+179}$ | 23.77 | | K ₇₃ | $s_{105} \Rightarrow s_{192+145}$ | 20.94 |
| K ₇₄ | $s_{106} \Rightarrow s_{192+181}$ | 18.24 | | K_{75} | $s_{107} \Rightarrow s_{192+182}$ | 19.97 |
| K ₇₆ | $s_{108} \Rightarrow s_{192+183}$ | 21.81 | | K ₇₇ | $s_{109} \Rightarrow s_{192+6}$ | 20.86 |
| K_{78} | $s_{110} \Rightarrow s_{192+7}$ | 22.83 |] | K_{79} | $s_{111} \Rightarrow s_{192+8}$ | 20.94 |

Table A.2: Number of IVs required to recover the key bits (80-bit IV)

Appendix B

Test Vectors of HC-256 and HC-128

The first 512 bits of keystream are given as test vectors for different values of key and IV. Note that for each 32-bit output given below, the least significant byte comes before the most significant byte in the keystream. For example, if S and T are 32-bit words, and $S = s_3||s_2||s_1||s_0$, $T = t_3||t_2||t_1||t_0$, where each s_i and t_i is one byte, and s_0 and t_0 denote the least significant bytes, then the keystream S, T is related to the keystream $s_0, s_1, s_2, s_3, t_0, t_1, t_2, t_3$.

B.1 Test Vectors of HC-256

Let $K = K_0 ||K_1|| \cdots ||K_7$ and $IV = IV_0 ||IV_1|| \cdots ||IV_7$.

1. The key and IV are set as 0.

| 8589075b | 0df3f6d8 | 2fc0c542 | 5179b6a6 |
|----------|----------|----------|----------|
| 3465f053 | f2891f80 | 8b24744e | 18480b72 |
| ec2792cd | bf4dcfeb | 7769bf8d | fa14aee4 |
| 7b4c50e8 | eaf3a9c8 | f506016c | 81697e32 |
| | | | |

2. The key is set as 0, the IV is set as 0 except that $IV_0 = 1$.

| e9ce174f | 8b05c2fe | b18bb1d1 |
|----------|--|--|
| 01312b71 | c61f50dd | 502a080b |
| 633d9241 | a6dac448 | af8561ff |
| 9448c434 | 2de7e9f3 | 37520bdf |
| | e9ce174f 01312b71 633d9241 9448c434 | e9ce174f8b05c2fe01312b71c61f50dd633d9241a6dac4489448c4342de7e9f3 |

3. The IV is set as 0, the key is set as 0 except that $K_0 = 0x55$.

| fe4a401c | ed5fe24f | d19a8f95 | 6fc036ae |
|----------|----------|----------|----------|
| 3c5aa688 | 23e2abc0 | 2f90b3ae | a8d30e42 |
| 59f03a6c | 6e39eb44 | 8f7579fb | 70137a5e |
| 6d10b7d8 | add0f7cd | 723423da | f575dde6 |

4. Let $A_i = \bigoplus_{j=0}^{0x \text{ffff}} s_{16j+i}$ for $i = 0, 1, \dots, 15$, i.e. set a 512-bit buffer as 0 and encrypt it repeatedly for 2^{20} times. Set the key and IV as 0, the value of $A_0 ||A_1|| \cdots ||A_{15}$ is given below:

| c6b6fb99 | f2ae1440 | a7d4ca34 | 2011694e |
|----------|----------|----------|----------|
| 6f36b4be | 420db05d | 4745fd90 | 7c630695 |
| 5f1d7bda | 13ae7e36 | aebc5399 | 733b7f37 |
| 95f34066 | b601d21f | 2d8cf830 | a9c08937 |

B.2 Test Vectors of HC-128

Let $K = K_0 ||K_1||K_2||K_3$ and $IV = IV_0 ||IV_1||IV_2||IV_3$.

1. The key and IV are set as 0.

731500823bfd03a0fb2fd77faa63af0ede122fc6a7dc29b662a685278b75ec689036db1e8189600500ade078491fbf9a1cdc30136c3d6e2490f664b29cd57102

2. The key is set as 0, the IV is set as 0 except that $IV_0 = 1$.

c01893d5b7dbe9588f65ec986417660436fc6724c82c6eec1b1c38a7c9b42a95323ef1230a6a908bce757b689f14f7bbe4cde011aeb5173f89608c94b5cf46ca

3. The IV is set as 0, the key is set as 0 except that $K_0 = 0x55$.

04b4930a b02af931 0639f032 518251a4 bcb4a47a 5722480b 2bf99f72 cdc0e566 310f0c56 d3cc83e8 663db8ef 62dfe07f 593e1790 c5ceaa9c ab03806f c9a6e5a0

4. Set the key and IV as 0, the value of $A_0||A_1||\cdots||A_{15}$ is:
| a4eac026 | 7e491126 | 6a2a384f | 5c4e1329 |
|----------|----------|----------|----------|
| da407fa1 | 55e6b1ae | 05c6fdf3 | bbdc8a86 |
| 7a699aa0 | 1a4dc117 | 63658ccc | d3e62474 |
| 9cf8236f | 0131be21 | c3a51de9 | d12290de |

List of Publications

- Hongjun Wu. "The Stream Cipher HC-128." New Stream Cipher Designs

 The eSTREAM Finalists, Lecture Notes in Computer Science 4986, M. Robshaw and O. Billet (Eds.), pp. 39-47, Springer-Verlag, 2008.
- Hongjun Wu and Bart Preneel. "Differential-Linear Attacks against the Stream Cipher Phelix." Fast Software Encryption – FSE 2007, Lecture Notes in Computer Science 4593, A. Biryukov (Ed.), pp. 87-100, Springer-Verlag, 2007.
- Hongjun Wu and Bart Preneel. "Differential Cryptanalysis of the Stream Ciphers Py, Py6 and Pypy." Advances in Cryptology – Eurocrypt 2007, Lecture Notes in Computer Science 4515, M. Naor (Ed.), pp. 276-290, Springer-Verlag, 2007.
- Hirotaka Yoshida, Dai Watanabe, Katsuyuki Okeya, Jun Kitahara, Hongjun Wu, Ozgul Kucuk and Bart Preneel. "MAME: A compression function proposal for RFID applications." Cryptographic Hardware and Embedded Systems – CHES 2007, Lecture Notes in Computer Science 4727, P. Paillier and I. Verbauwhede (Eds.), pp. 148-165, Springer-Verlag, 2007.
- Hongjun Wu and Bart Preneel. "Cryptanalysis of the Stream Cipher ABC v2." Selected Areas in Cryptography – SAC 2006, Lecture Notes in Computer Science 4356, E. Biham and A. M. Youssef (Eds.), pp. 56-66, Springer-Verlag, 2006.
- Hongjun Wu and Bart Preneel. "Resynchronization Attacks on WG and LEX." Fast Software Encryption – FSE 2006, Lecture Notes in Computer Science 4047, M. J. Robshaw (Ed.), pp. 422-432, Springer-Verlag, 2006.
- Hongjun Wu and Bart Preneel. "Cryptanalysis of the Stream Cipher DECIM." Fast Software Encryption – FSE 2006, Lecture Notes in Computer Science 4047, M. J. Robshaw (Ed.), pp. 30-40, Springer-Verlag, 2006.

127

- Bin Zhang, Hongjun Wu, Dengguo Feng and Feng Bao. "A Fast Correlation Attack on the Shrinking Generator." RSA Conference – Cryptographers' Track (CT-RSA 2005), Lecture Notes in Computer Science 3376, pp. 72-86, Springer-Verlag, 2004.
- Bin Zhang, Hongjun Wu, Dengguo Feng and Hong Wang. "Weaknesses of COSvd (2, 128) Stream Cipher." International Conference on Information Security and Cryptology – ICISC 2005, Lecture Notes in Computer Science 3935, D. Won and S. Kim (Eds.), pp. 270-283, Springer-Verlag, 2006.
- Hongjun Wu. "A New Stream Cipher HC-256." In Fast Software Encryption – FSE 2004, Lecture Notes in Computer Science 3017, B. K. Roy and W. Meier (Eds.), pp. 226-244, Springer-Verlag, 2004.
- Hongjun Wu and Di Ma. "Efficient and Secure Encryption Schemes for JPEG2000." International Conference on Acoustics, Speech, and Signal Processing – ICASSP 2004. MSP-L 1.6, Vol. V, pp. 869-872.
- Bin Zhang, Hongjun Wu, Dengguo Feng and Feng Bao. "Security Analysis of the Generalized Self-Shrinking Generator." International Conference on Information and Communications Security – ICICS 2004, Lecture Notes in Computer Science 3269, J. Lopez, S. Qing and E. Okamoto (Eds.), pp. 388-400, Springer-Verlag, 2004.
- Bin Zhang, Hongjun Wu, Dengguo Feng and Feng Bao. "Chosen Ciphertext Attack on a New Class of Self-Synchronizing Stream Ciphers." Progress in Cryptology – INDOCRYPT 2004, Lecture Notes in Computer Science 3348, A. Canteaut and K. Viswanathan (Eds.), pp. 73-83, Springer-Verlag, 2005.
- Bin Zhang, Hongjun Wu, Feng Bao and Dengguo Feng. "Cryptanalysis of a Knapsack Based Two-Lock Cryptosystem." Applied Cryptography and Network Security – ACNS 2004, Lecture Notes in Computer Science 3089, M. Jakobsson, M. Yung and J. Zhou (Eds.), pp. 303-309, Springer-Verlag, 2004.
- Hongjun Wu, Feng Bao and Robert H. Deng. "Improved Known Plaintext Attack on FEA-M." International Conference on Information and Communication Security – ICICS 2003, Lecture Notes in Computer Science 2836, S. Qing, D. Gollmann and J. Zhou (Eds.), pp. 84-87, Springer-Verlag, 2003.
- Hongjun Wu and Feng Bao. "Cryptanalysis of Stream Cipher COS(2, 128) Mode I." Australasian Conference on Information Security and Privacy – ACISP 2002, Lecture Notes in Computer Science 2384, L. Batten and J. Seberry (Eds.), pp. 154-158, Springer-Verlag, 2002.

- Hongjun Wu. "Cryptanalysis of Stream Cipher Alpha1." Australasian Conference on Information Security and Privacy – ACISP 2002, Lecture Notes in Computer Science 2384, L. Batten and J. Seberry (Eds.), pp. 169-175, Springer-Verlag, 2002.
- Hongjun Wu. "Related-Cipher Attacks." International Conference on Information and Communication Security – ICICS 2002, Lecture Notes in Computer Science 2513, R. Deng, S. Qing, F. Bao and J. Zhou (Eds.), pp. 447-455, Springer-Verlag, 2002.
- Hongjun Wu, Feng Bao and Robert H. Deng. "Cryptanalysis of Some Hash Functions Based on Block Ciphers and Codes." Informatica (Slovenia) 26(3): (2002).
- 20. Feng Bao, Robert H. Deng, Peirong Feng, Yan Guo and Hongjun Wu, "Secure and Private Distribution of Online Video and Some Related Cryptographic Issues." Australasian Conference on Information Security and Privacy – ACISP 2001, Lecture Notes in Computer Science 2119, V. Varadharajan and Y. Mu (Eds.), pp. 190-205, Springer-Verlag, 2001.
- Feng Bao, Robert H. Deng, Willi Geiselmann, Claus-Peter Schnorr, Rainer Steinwandt and Hongjun Wu, "Cryptanalysis of Two Sparse Polynomial Based Public Key Cryptosystems." Public Key Cryptography – PKC 2001, Lecture Notes in Computer Science 1992, K. Kim (Ed.), pp. 153-164, Springer-Verlag, 2001.
- 22. Hongjun Wu, Feng Bao and Robert H. Deng, "Cryptanalysis of a Digital Signature Scheme on ID-Based Key-Sharing Infrastructures." Public Key Cryptography – PKC 2001, Lecture Notes in Computer Science 1992, K. Kim (Ed.), pp. 173-179, Springer-Verlag, 2001.
- Hongjun Wu, Feng Bao, Ding-Feng Ye and Robert H. Deng. "Cryptanalysis of Polynominal Authentication and Signature Scheme." Australasian Conference on Information Security and Privacy – ACISP 2000, Lecture Notes in Computer Science 1841, E. Dawson, A. Clark and C. Boyd (Eds.), pp. 278-288, Springer-Verlag, 2000.
- Hongjun Wu, Feng Bao, Ding-Feng Ye and Robert H. Deng, "Cryptanalysis of the m-Permutation Protection Schemes." Australasian Conference on Information Security and Privacy – ACISP 2000, Lecture Notes in Computer Science 1841, E. Dawson, A. Clark and C. Boyd (Eds.), pp. 97-111, Springer-Verlag, 2000.

129

- Hongjun Wu, Feng Bao, Robert H. Deng and Qin-Zhong Ye. "Cryptanalysis of Rijmen-Preneel Trapdoor Ciphers." Advances in Cryptology – ASIACRYPT'98, Lecture Notes in Computer Science 1514, K. Ohta and D. Pei (Eds.), pp. 126-132, Springer-Verlag, 1998.
- Hongjun Wu, Feng Bao, Robert H. Deng and Qin-Zhong Ye. "Improved Truncated Differential Attacks on SAFER." Advances in Cryptology – ASI-ACRYPT'98, Lecture Notes in Computer Science 1514, K. Ohta and D. Pei (Eds.), pp. 133-147, Springer-Verlag, 1998.

Other Articles

- 27. Hongjun Wu. "The Misuse of RC4 in Microsoft Word and Excel." IACR ePrint, 2005. Available at http://eprint.iacr.org/2005/007
- 28. Hongjun Wu. "Breaking the Stream Cipher Whitenoise." IACR ePrint, 2003. Available at http://eprint.iacr.org/2003/250

Hongjun Wu received the Bachelor and Master of Engineering degrees from the Department of Electrical Engineering of the National University of Singapore in 1998 and 2000, respectively. From 1999 to 2005, he was working on information security at the Institute for Infocomm Research in Singapore. In 2005, he started doctoral studies in the research group COSIC (Computer Security and Industrial Cryptography) at the Department of Electrical Engineering (ESAT) of the Katholieke Universiteit Leuven in Belgium.